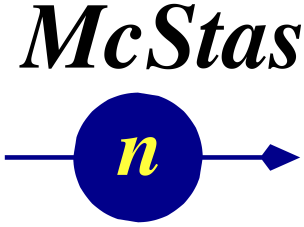


Physics Physics Department,
Technical University of Denmark
2800 Kongens Lyngby, Denmark



User and Programmers Guide to the Neutron Ray-Tracing Package McStas, version 3.5

P. Willendrup, E. Farhi, K. Lefmann

November, 2026

The software package McStas is a tool for carrying out Monte Carlo ray-tracing simulations of neutron scattering instruments with high complexity and precision. The simulations can compute all aspects of the performance of instruments and can thus be used to optimize the use of existing equipment, design new instrumentation, and carry out virtual experiments for e.g. training, experimental planning or data analysis. McStas is based on a unique design where an automatic compilation process translates high-level textual instrument descriptions into efficient ISO-C code. This design makes it simple to set up typical simulations and also gives essentially unlimited freedom to handle more unusual cases.

This report constitutes the reference manual for McStas, and, together with the manual for the McStas components, it contains documentation of most aspects of the program. It covers the various ways to compile and run simulations, a description of the meta-language used to define simulations, and some example simulations performed with the program.

This report documents McStas version 3.5, released November, 2026

The authors are:

Peter Kjær Willendrup <pkwi@fysik.dtu.dk>
Physics Department, Technical University of Denmark, Kongens Lyngby, Denmark

Emmanuel Farhi <emmanuel.farhi@synchrotron-soleil.fr>
Synchrotron SOLEIL, Saint-Aubin, France

Kim Lefmann <lefmann@nbi.dk>
Niels Bohr Institute, University of Copenhagen, Denmark

as well as authors who left the project:

Erik Knudsen <erkn@fysik.dtu.dk>
Physics Department, Technical University of Denmark, Kongens Lyngby, Denmark Jakob Garde <jaga@fysik.dtu.dk>
Physics Department, Technical University of Denmark, Kongens Lyngby, Denmark Jonas Stein <Jonas.Stein@uni-koeln.de>
Institute of Physics II, University of Cologne, Germany Peter Christiansen <pchristi@hep.lu.se>
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Present address: University of Lund, Lund, Sweden
Klaus Lieutenant <k.lieutenant@fz-juelich.de>
Institut Laue-Langevin, Grenoble, France
Present address: Forschungs-Zentrum Jülich, Germany
Kristian Nielsen <kristian-nielsen@mail.tele.dk>
Materials Research Department, Risø National Laboratory, Roskilde, Denmark
Presently associated with: MySQL AB, Sweden

ISBN 978–87–550–3679–6
ISSN 0106–2840

Physics Department · DTU · 2026

Contents

Preface and acknowledgments **10**

1. Introduction to McStas **11**
1.1. Development of Monte Carlo neutron simulation . . . 11
1.2. Scientific background . . . 12
1.2.1. The goals of McStas . . . 14
1.3. The design of McStas . . . 14
1.4. Overview . . . 16

2. Installing McStas **18**

3. Monte Carlo Techniques and simulation strategy **19**
3.1. Neutron spectrometer simulations . . . 19
3.1.1. Monte Carlo ray tracing simulations . . . 19
3.2. The neutron weight . . . 20
3.2.1. Statistical errors of non-integer counts . . . 20
3.3. Weight factor transformations during a Monte Carlo choice . . . 21
3.3.1. Direction focusing . . . 22
3.4. Adaptive and Stratified sampling . . . 22
3.5. Accuracy of Monte Carlo simulations . . . 23

4. Running McStas **25**
4.1. Installation and updates . . . 25
4.1.1. Important note for Windows users . . . 27
4.1.2. New releases of McStas . . . 27
4.2. Brief introduction to the graphical user interface . . . 27
4.3. Running the instrument compiler . . . 29
4.3.1. Code generation options . . . 29
4.3.2. Specifying the location of files . . . 30
4.3.3. Embedding the generated simulations in other programs . . . 31
4.3.4. Running the C compiler . . . 32
4.4. Running the simulations . . . 33
4.4.1. Choosing an output data file format . . . 34
4.4.2. Basic import and plot of results . . . 35
4.4.3. Interacting with a running simulation . . . 38
4.4.4. Optimizing simulation speed . . . 39
4.4.5. Optimizing instrument parameters . . . 40

4.5. Using simulation front-ends . . . 42
4.5.1. The graphical user interface (mcgui) . . . 43
4.5.2. Running simulations on the commandline (mcrun) . . . 49
4.5.3. Interactive notebooks (mcstas-jupylab) . . . 51
4.5.4. GPU acceleration via OpenACC . . . 51
4.5.5. Graphical display of simulations (mcdisplay) . . . 52
4.5.6. Plotting the results of a simulation (mcplot) . . . 54
4.5.7. Plotting resolution functions (mcresplot) . . . 56
4.5.8. Creating and viewing the library, component/instrument help and Manuals (mcdoc) . . . 57
4.5.9. Self-testing the installation (mctest, mcviewtest) . . . 57
4.6. Data formats - Analyzing and visualizing the simulation results . . . 59
4.6.1. The McStas/McCode text format . . . 59
4.6.2. NeXus format . . . 60
4.7. Using MPI for parallel computing . . . 60
4.7.1. Parallel computing (MPI) . . . 61
4.7.2. McRun options for MPI . . . 62
4.7.3. McStas/MPI Performance . . . 63
4.7.4. MPI Bugs and limitations . . . 64

5. The McStas kernel and meta-language 65
5.1. Notational conventions . . . 65
5.2. Syntactical conventions . . . 66
5.3. Writing instrument definitions . . . 69
5.3.1. The instrument definition head . . . 69
5.3.2. The `DEPENDENCY` line . . . 70
5.3.3. The `SEARCH` line . . . 70
5.3.4. The `SHELL` line . . . 71
5.3.5. The `DECLARE` section . . . 71
5.3.6. The `USERVARS` section . . . 71
5.3.7. The `INITIALIZE` section . . . 72
5.3.8. The `NEXUS` extension . . . 72
5.3.9. The `TRACE` section . . . 73
5.3.10. The `SAVE` section . . . 74
5.3.11. The `FINALLY` section . . . 75
5.3.12. The end of the instrument definition . . . 75
5.3.13. Code for the instrument `vanadium_example.instr` . . . 75
5.4. Writing instrument definitions - complex arrangements and syntax . . . 75
5.4.1. Embedding instruments in instruments TRACE . . . 76
5.4.2. Groups and component extensions - GROUP - EXTEND . . . 76
5.4.3. Duplication of component instances - COPY . . . 78
5.4.4. Conditional components - WHEN . . . 79
5.4.5. Component loops and non sequential propagation - JUMP . . . 80
5.4.6. Enhancing statistics reaching components - SPLIT . . . 81

5.4.7. Attaching metadata to components and instruments - METADATA 82
5.5. Writing component definitions . . . 83
5.5.1. The component definition header . . . 83
5.5.2. The `NOACC` flag . . . 85
5.5.3. The `DEPENDENCY` line . . . 85
5.5.4. The `DECLARE` section . . . 85
5.5.5. The `SHARE` section . . . 86
5.5.6. The `INITIALIZE` section . . . 87
5.5.7. The `TRACE` section . . . 87
5.5.8. The `SAVE` section . . . 88
5.5.9. The `FINALLY` section . . . 90
5.5.10. The `MCDISPLAY` section . . . 91
5.5.11. The end of the component definition . . . 92
5.5.12. A component example: Slit . . . 92
5.6. Extending component definitions . . . 92
5.6.1. Extending from the instrument definition . . . 93
5.6.2. Component heritage and duplication . . . 93
5.7. McDoc, the McStas library documentation tool . . . 94
5.7.1. Documentation generators mcdoc and mcgui . . . 94
5.7.2. The format of the comments in the library source code . . . 94

6. Polarisation in McStas 96
6.1. The Polarization Vector . . . 96
6.1.1. Example: Magnetic fields . . . 98
6.2. Polarized Neutron Scattering . . . 99
6.2.1. Example: Nuclear scattering . . . 100
6.2.2. Example: Polarizing Monochromator and Guides . . . 101
6.3. Magnetic fields: the precession algorithm and field framework . . . 103
6.3.1. The magnetic field stack . . . 103
6.3.2. Numerical spin precession: `SimpleNumMagnetPrecession` . . . 104
6.3.3. Available field functions . . . 105
6.4. Polarisation components in McStas . . . 106
6.4.1. Setting and analysing polarisation . . . 106
6.4.2. Polarising monochromators, mirrors and guides . . . 107
6.4.3. Magnetic field components . . . 107
6.4.4. Polarisation-sensitive monitors . . . 108
6.4.5. Samples and polarisation . . . 108
6.5. Test and example instruments . . . 109
6.5.1. Component-level regression tests . . . 109
6.5.2. Test_Pol_TripleAxis . . . 109
6.5.3. He3_spin_filter . . . 109
6.5.4. A spin-echo example: SE_example . . . 109

7. The Union components **111**
7.1. Motivation . . . 111
7.1.1. Volumes, priority and overlap . . . 112
7.2. Component classes . . . 112
7.2.1. Loading the Union library . . . 113
7.2.2. Scattering processes . . . 113
7.2.3. Union_make_material . . . 115
7.2.4. Geometry components . . . 116
7.2.5. Union_master . . . 118
7.2.6. Surfaces . . . 119
7.2.7. Logger components . . . 119
7.2.8. Conditional components . . . 120
7.2.9. Tagging: ray histories . . . 120
7.3. The propagation algorithm . . . 121
7.3.1. Basic Monte Carlo step . . . 121
7.3.2. Intersection and destination lists . . . 121
7.3.3. Known limitations . . . 122
7.4. A complete example . . . 123
7.5. Further examples and worked tutorials . . . 124

8. Links to other computing codes **125**
8.1. McStas and MANTID . . . 125
8.1.1. System requirements . . . 125
8.1.2. Requirements for the instrument file . . . 125
8.1.3. Compiling and running your simulation for Mantid output . . . 126
8.1.4. Looking at instrument output in Mantid . . . 127
8.2. McStas and MCNP(X) . . . 128

9. The component library: Abstract **129**
9.1. Component categories . . . 129
9.2. Data files . . . 130
9.3. Component and instrument examples . . . 130

10. Instrument examples **132**
10.1. A quick tour of instrument examples . . . 132
10.1.1. Templates . . . 132
10.1.2. Risoe . . . 132
10.1.3. ISIS . . . 133
10.1.4. ILL . . . 133
10.1.5. ESS . . . 133
10.1.6. Union_demos, Union_sample_environments, Union_validation . . . 133
10.1.7. NCrystal . . . 133
10.1.8. Tests_* categories . . . 134
10.1.9. Other facility categories . . . 134

10.2. A test instrument for the component V_sample . . . 134
10.2.1. Scattering from the V-sample test instrument . . . 135
10.3. The triple axis spectrometer TAS1 . . . 135
10.3.1. Simulated and measured resolution of TAS1 . . . 137
10.4. The time-of-flight spectrometer PRISMA . . . 140
10.4.1. Simple spectra from the PRISMA instrument . . . 141

A. Random numbers in McStas **142**
A.1. Transformation of random numbers . . . 142
A.2. Random generator . . . 143

B. Libraries and constants **144**
B.1. Run-time calls and functions (`mcstas-r`) . . . 144
B.1.1. Neutron propagation . . . 144
B.1.2. Coordinate and component variable retrieval . . . 145
B.1.3. Coordinate transformations . . . 147
B.1.4. Mathematical routines . . . 147
B.1.5. Output from detectors . . . 148
B.1.6. Ray-geometry intersections . . . 148
B.1.7. Random numbers . . . 148
B.2. Reading a data file into a vector/matrix (Table input, `read_table-lib`) . 149
B.3. Monitor_nD Library . . . 152
B.4. Adaptive importance sampling Library . . . 152
B.5. Vitess import/export Library . . . 152
B.6. Constants for unit conversion etc. . . . 152

C. The McStas terminology **154**

Bibliography **155**

Index and keywords **158**

Preface and acknowledgments

This document contains information on the Monte Carlo neutron ray-tracing program McStas version 3.5, building on the initial release in October 1998 of version 1.0 as presented in Ref. [LN99] and further developed though version 2.0 as presented in Ref. [Wil+14]. The reader of this document is supposed to have some knowledge of neutron scattering, whereas only little knowledge about simulation techniques is required. In a few places, we also assume familiarity with the use of the C programming language and UNIX/Linux.

If you don't want to read this manual in full, go directly to the brief introduction in chapter 4.2.

It is a pleasure to thank Prof. Kurt N. Clausen, PSI, for his continuous support to this project and for having initiated McStas in the first place. Essential support has also been given by Prof. Robert McGreevy, ISIS. We have also benefited from discussions with many other people in the neutron scattering community, too numerous to mention here.

In case of errors, questions, or suggestions, do not hesitate to contact the authors at `mcstas-users@mcstas.org` or consult the McStas home page [Mcsa]. A special bug/request reporting service is available [Git].

If you **appreciate** this software, please subscribe to the `mcstas-users@mcstas.org` email list, send us a smiley message, and contribute to the package. We also encourage you to refer to this software when publishing results, with the following citations:

- P. Willendrup, and K. Lefmann, Journal of Neutron Research, **23** (2021) 7-27.
- P. Willendrup, and K. Lefmann, Journal of Neutron Research, **22**, (2020) 1-16.
- P. Willendrup, E. Farhi, E. Knudsen, U. Filges and K. Lefmann, Journal of Neutron Research, **17** (2014) 35.
- P. Willendrup, E. Farhi and K. Lefmann, Physica B, **350** (2004) 735.
- K. Lefmann and K. Nielsen, Neutron News **10/3**, 20, (1999).

The McStas project has been supported by the European Union through"XENNI / Cool Neutrons" (FP4), "SCANS" (FP5), "nmi3/MCNSI" (FP6), "'nmi3-ii/E-learning" and "nmi3-ii/MCNSI7" (FP7) [Nmi; Mcn]. McStas was supported directly from the construction project for the ISIS second target station (TS2/EU), see [Ts2]. Currently McStas is supported through the Danish involvement in the *Data Management and Software Center*, a subdivision of the European Spallation Source (ESS), see [Ess] and the European Union through "SINE2020/WP3 e-elarning" and and "SINE2020/WP8 e-Tools" (Horizon2020). the home pages [Sin].

1. Introduction to McStas

Efficient design and optimization of neutron spectrometers are formidable challenges, which are efficiently treated by Monte Carlo simulation techniques. When McStas version 1.0 was released in October 1998, except for the NISP/MCLib program [Nis], no existing package offered a general framework for the neutron scattering community to tackle the problems currently faced at reactor and spallation sources. The McStas project was designed to provide such a framework.

McStas is a fast and versatile software tool for neutron ray-tracing simulations. It is based on a meta-language specially designed for neutron simulation. Specifications are written in this language by users and automatically translated into efficient simulation codes in ISO-C. The present version supports both continuous and pulsed source instruments, and includes a library of standard components with in total around 130 components. These enable to simulate all kinds of neutron scattering instruments (diffractometers, spectrometers, reflectometers, small-angle, back-scattering,...) for both continuous and pulsed sources.

The core McStas package is written in ISO-C, with various tools based on Perl and Python and is freely available for download from the McStas website [Mcsa]. The package is actively being developed and supported by DTU Physics, Institut Laue Langevin (ILL), Paul Scherrer Institute and the Niels Bohr Institute (NBI). The system is well tested and is supplied with several examples and with an extensive documentation. Besides this manual, a separate component manual exists.

1.1. Development of Monte Carlo neutron simulation

The very early implementations of the method for neutron instruments used *home-made* computer programs (see e.g. papers by J.R.D. Copley, D.F.R. Mildner, J.M. Carpenter, J. Cook), more general packages have been designed, providing models for most parts of the simulations. These present existing packages are: NISP [See+00], ResTrax [SK97], McStas [Wil+14; LN99; WFL04; Wil+14; Mcsa], Vitess [Wec+00; Vit], IDEAS [LW02] and IB (Instrument Builder) [Ibw]. Supplementing the Monte Carlo based methods, various analytic phase-space simulation methods exist, including Neutron Acceptance Diagram Shading (NADS) [Nad]. Their usage usually covers all types of neutron spectrometers, most of the time through a user-friendly graphical interface, without requiring programming skills.

The neutron ray-tracing Monte-Carlo method has been used widely for *e.g.* guide studies [Cop93; Far+02; Sch+04], instrument optimization and design [ZLa04; Lie05]. Most of the time, the conclusions and general behavior of such studies may be obtained using

the classical analytic approaches, but accurate estimates for the flux, the resolutions, and generally the optimum parameter set, benefit advantageously from MC methods.

Recently, the concept of virtual experiments, *i.e.* full simulations of a complete neutron experiment, has been suggested as a major asset for neutron ray-tracing simulations. The goal is that simulations should be of benefit to not only instrument builders, but also to users for training, experiment planning, diagnostics, and data analysis.

In the late 90'ies at Risø National Laboratory, simulation tools were urgently needed, not only to better utilize existing instruments (*e.g.* RITA-1 and RITA-2 [Mas+95; Cla+98; Lef+00]), but also to plan completely new instruments for new sources (*e.g.* the Spallation Neutron Source, SNS [Sns] and the planned European Spallation Source, ESS [Ess]). Writing programs in C or FORTRAN for each of the different cases involves a huge effort, with debugging presenting particularly difficult problems. A higher level tool specially designed for simulating neutron instruments was needed. As there was no existing simulation software that would fulfill our needs, the McStas project was initiated. In addition, the ILL required an efficient and general simulation package in order to achieve renewal of its instruments and guides. A significant contribution to both the component library and the McStas kernel itself was early performed at the ILL and included in the package. ILL later became a part of the core McStas team. Similarly, the PSI has applied McStas extensively for instrument design and upgrades, provided important component additions and contributed several systematic comparative studies of the European instrument Monte Carlo codes. Hence, PSI has also become a part of the core McStas team. Since year 2001 Risø was no longer a neutron source, and the authors from that site have moved on to positions at University of Copenhagen (NBI) and Technical University of Denmark (DTU Physics), hence these two partners have joined the core McStas team. Finally, trough general emphasis on use of McStas as a tool for simulating the ESS instruments and virtual data, plus the partial secondment of one DTU-based McStas author, the ESS Data Management and Software Centre (ESS DMSC) is now contributing to the projecte.

1.2. Scientific background

What makes scientists happy? Probably collect good quality data, pushing the instruments to their limits, and fit that data to physical models. Among available measurement techniques, neutron scattering provides a large variety of spectrometers to probe structure and dynamics of all kinds of materials.

Neutron scattering instruments are built as a series of neutron optics elements. Each of these elements modifies the beam characteristics (e.g. divergence, wavelength spread, spatial and time distributions) in a way which, for simple neutron beam configurations, may be modeled with analytic methods. This is valid for individual elements such as guides [MLS63; Mil90], choppers [Low60; Cop03], Fermi choppers [FMM47; Pet05], velocity selectors [Cla+66], monochromators [Fre83; Sea97; SST02; Ali04], and detectors [Rad74; Pes+89; Man+04]. In the case of a limited number of optical elements, the so-called acceptance diagram theory [Mil90; Cop93; Cus03] may be used, within which the

neutron beam distributions are considered to be homogeneous, triangular or Gaussian. However, real neutron instruments are constituted of a large number of optical elements, and this brings additional complexity by introducing strong correlations between neutron beam parameters like divergence and position - which is the basis of the acceptance diagram method - but also wavelength and time. The usual analytic methods, such as phase-space theory, then reach their limit of validity in the description of the resulting effects.

In order to cope with this difficulty, Monte Carlo (MC) methods (for a general review, see Ref. [Jam80]) may be applied to the simulation of neutron instruments. The use of probability is common place in the description of microscopic physical processes. Integrating these events (absorption, scattering, reflection, ...) over the neutron trajectories results in an estimation of measurable quantities characterizing the neutron instrument. Moreover, using variance reduction (importance sampling) where possible, reduces the computation time and gives better accuracy.

Early implementations of the MC method for neutron instruments used *home-made* computer programs (see [Cop+86; MPC77]) but, more recently, general packages have been designed, providing models for most optical components of neutron spectrometers. The most widely-used packages are NISP [See+00], ResTrax [SK97], McStas [Wil+14; LN99; Mcsa], Vitess [Wec+00], and IDEAS [LW02], which allow a wide range of neutron scattering instruments to be simulated.

The neutron ray-tracing Monte Carlo method has been used widely for guide studies [Cop93; Far+02; Sch+04], instrument optimization and design [ZLa04; Lie05]. Most of the time, the conclusions and general behavior of such studies may be obtained using the classical analytic approaches, but accurate estimates for the flux, resolution and generally the optimum parameter set, benefit considerably from MC methods, see Chapter 3.

Neutron instrument resolution (in q and E) and flux are often limitations in the experiments. This then motivates instrument responsibles to improve the resolution, flux and overall efficiency at the spectrometer positions, and even to design new machines. Using both analytic and numerical methods, optimal configurations may be found.

But achieving a satisfactory experiment on the best neutron spectrometer is not all. Once collected, the data analysis process raises some questions concerning the signal: what is the background signal? What proportion of coherent and incoherent scattering has been measured? Is possible to identify clearly the purely elastic (structure) contribution from the quasi-elastic and inelastic one (dynamics)? What are the contributions from the sample geometry, the container, the sample environment, and generally the instrument itself? And last but not least, how does multiple scattering affect the signal? Most of the time, the physicist will elude these questions using rough approximations, or applying analytic corrections [Cop+86]. Monte-Carlo techniques also provide means to evaluate some of these quantities.

Technicalities of Monte-Carlo simulation techniques are explained in detail in Chapter 3.

1.2.1. The goals of McStas

Initially, the McStas project had four main objectives that determined its design.

Correctness. It is essential to minimize the potential for bugs in computer simulations. If a word processing program contains bugs, it will produce bad-looking output or may even crash. This is a nuisance, but at least you know that something is wrong. However, if a simulation contains bugs it produces wrong results, and unless the results are far off, you may not know about it! Complex simulations involve hundreds or even thousands of lines of formulae, making debugging a major issue. Thus the system should be designed from the start to help minimize the potential for bugs to be introduced in the first place, and provide good tools for testing to maximize the chances of finding existing bugs.

Flexibility. When you commit yourself to using a tool for an important project, you need to know if the tool will satisfy not only your present, but also your future requirements. The tool must not have fundamental limitations that restrict its potential usage. Thus the McStas systems needs to be flexible enough to simulate different kinds of instruments as well as many different kind of optical components, and it must also be extensible so that future, as yet unforeseen, needs can be satisfied.

Power. "*Simple things should be simple; complex things should be possible*". New ideas should be easy to try out, and the time from thought to action should be as short as possible. If you are faced with the prospect of programming for two weeks before getting any results on a new idea, you will most likely drop it. Ideally, if you have a good idea at lunch time, the simulation should be running in the afternoon.

Efficiency. Monte Carlo simulations are computationally intensive, hardware capacities are finite (albeit impressive), and humans are impatient. Thus the system must assist in producing simulations that run as fast as possible, without placing unreasonable burdens on the user in order to achieve this.

1.3. The design of McStas

In order to meet these ambitious goals, it was decided that McStas should be based on its own *meta-language* (also known as *domain-specific language*), specially designed for simulating neutron scattering instruments. Simulations are written in this language by the user, and the McStas compiler automatically translates them into efficient simulation programs written in ISO-C.

In realizing the design of McStas, the task was separated into four conceptual layers, listed below and also detailed in Figure 4.1

1. Graphical user interface and scripting layer, presentation of the calculations, graphical or otherwise. (aka. the tool layer).

2. Modeling of the overall instrument geometry, mainly consisting of the type and position of the individual components.

3. Modeling the physical processes of neutron scattering, *i.e.* the calculation of the fate of a neutron that passes through the individual components of the instrument (absorption, scattering at a particular angle, etc.)

4. Accurate calculation, using Monte Carlo techniques, of instrument properties such as resolution function from the result of ray-tracing of a large number of neutrons. This includes estimating the accuracy of the calculation.

If you don't want to read this manual in full, go directly to the brief introduction in chapter 4.2.

Though obviously interrelated, these four layers can be treated independently, and this is reflected in the overall system architecture of McStas. The user will in many situations be interested in knowing the details only in some of the layers. For example, one user may merely look at some results prepared by others, without worrying about the details of the calculation. Another user may simulate a new instrument without having to reinvent the code for simulating the individual components in the instrument. A third user may write an intricate simulation of a complex component, e.g. a detailed description of a rotating velocity selector, and expect other users to easily benefit from his/her work, and so on. McStas attempts to make it possible to work at any combination of layers in isolation by separating the layers as much as possible in the design of the system and in the meta-language in which simulations are written.

The usage of a special meta-language and an automatic compiler has several advantages over writing a big monolithic program or a set of library functions in C, FORTRAN, or another general-purpose programming language. The meta-language is more *powerful*; specifications are much simpler to write and easier to read when the syntax of the specification language reflects the problem domain. For example, the geometry of instruments would be much more complex if it were specified in C code with static arrays and pointers. The compiler can also take care of the low-level details of interfacing the various parts of the specification with the underlying C implementation language and each other. This way, users do not need to know about McStas internals to write new component or instrument definitions, and even if those internals change in later versions of McStas, existing definitions can be used without modification.

The McStas system also utilizes the meta-language to let the McStas compiler generate as much code as possible automatically, letting the compiler handle some of the things that would otherwise be the task of the user/programmer. *Correctness* is improved by having a well-tested compiler generate code that would otherwise need to be specially written and debugged by the user for every instrument or component. *Efficiency* is also improved by letting the compiler optimize the generated code in ways that would be time-consuming or difficult for humans to do. Furthermore, the compiler can generate several different simulations from the same specification, for example to optimize the simulations in different ways, to generate a simulation that graphically displays neutron

trajectories, and possibly other things in the future that were not even considered when the original instrument specification was written.

The design of McStas makes it well suited for doing "what if..." types of simulations. Once an instrument has been defined, questions such as "what if a slit was inserted", "what if a focusing monochromator was used instead of a flat one", "what if the sample was offset 2 mm from the center of the axis" and so on are easy to answer. Within minutes the instrument definition can be modified and a new simulation program generated. It also makes it simple to debug new components. A test instrument definition may be written containing a neutron source, the component to be tested, and whatever monitors are useful, and the component can be thoroughly tested before being used in a complex simulation with many different components.

The McStas system is based on ISO-C, making it both efficient and portable. The meta-language allows the user to embed arbitrary C code in the specifications. *Flexibility* is thus ensured since the full power of the C language is available if needed.

1.4. Overview

The McStas system documentation consists of the following major parts:

- The complete listing of changes related to the version McStas 3.5 is available in CHANGES document of the relevant download folder at `https://download.mcstas.org/`. Bugs are reported and traced using the McStas GitHub issue system[Git]. We will not present here an extensive list of corrections, and we let the reader refer to this bug reporting service for details. Only important changes are indicated in the CHANGES document. In addition to this manual, extensive and up-to-date documentation is maintained on the McStas/McCode GitHub wiki [Wik]:

  `https://github.com/mccode-dev/McCode/wiki`

  The wiki covers end-user tutorials, component development guides, GPU acceleration, and migration from McStas 2.x to 3.x.

- Chapter 3 concerns Monte Carlo techniques and simulation strategies in general

- Chapter 4 includes a brief introduction to the McStas system (section 4.2) as well a section (4.3) on running the compiler to produce simulations. Section 4.4 explains how to run the generated simulations. Running McStas on parallel computers require special attention and is discussed in section 4.7. A number of front-end programs are used to run the simulations and to aid in the data collection and analysis of the results. These user interfaces are described in section 4.5.

- The McStas meta-language is described in chapter 5. This chapter also describes a set of library functions and definitions that aid in the writing of simulations. See appendix B for more details.

- The McStas component library contains a collection of well-tested, as well as user contributed, beam components that can be used in simulations. The McStas component library is documented in a separate manual and on the McStas web-page [Mcsa], but a short overview of these components is given in chapter 9 of the Manual.

As of this release of McStas support for simulating neutron polarization is strongly improved, e.g. by allowing nested magnetic fields, tabulated magnetic fields in numerical input files and by close to "full" support of polarization in all components. As this is the first stable release with these new features, functionality is likely to change. To reflect this, the documentation is still only available in the appendix of the Component manual. A list of library calls that may be used in component definitions appears in appendix B, and an explanation of the McStas terminology can be found in appendix C of the Manual..

2. Installing McStas

Up to date installation instructions for all supported platforms (conda, Linux/Debian, macOS, Windows) are maintained on GitHub at `https://github.com/mccode-dev/McCode/blob/main/INSTALL-McStas/README.md`

3. Monte Carlo Techniques and simulation strategy

This chapter explains the simulation strategy and the Monte Carlo techniques used in McStas. We first explain the concept of the neutron weight factor, and discuss the statistical errors in dealing with sums of neutron weights. Secondly, we give an expression for how the weight factor transforms under a Monte Carlo choice and specialize this to the concept of direction focusing. Finally, we present a way of generating random numbers with arbitrary distributions. More details are available in the Appendix concerning random numbers.

3.1. Neutron spectrometer simulations

3.1.1. Monte Carlo ray tracing simulations

The behavior of a neutron scattering instrument can in principle be described by a complex integral over all relevant parameters, like initial neutron energy and divergence, scattering vector and position in the sample, etc. However, in most relevant cases, these integrals are not solvable analytically, and we hence turn to Monte Carlo methods. The neutron ray-tracing Monte Carlo method has been used widely for guide studies [Cop93; Far+02; Sch+04], instrument optimization and design [ZLa04; Lie05]. Most of the time, the conclusions and general behavior of such studies may be obtained using the classical analytic approaches, but accurate estimates for the flux, resolution and generally the optimum parameter set, benefit considerably from MC methods.

Mathematically, the Monte-Carlo method is an application of the law of large numbers [Jam80; GRR92]. Let $f(u)$ be a finite continuous integrable function of parameter u for which an integral estimate is desirable. The discrete statistical mean value of f (computed as a series) in the uniformly sampled interval $a < u < b$ converges to the mathematical mean value of f over the same interval.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1, a\le u_i\le b}^{n} f(u_i) = \frac{1}{b-a} \int_a^b f(u)du \tag{3.1}$$

In the case were the u_i values are regularly sampled, we come to the well known midpoint integration rule. In the case were the u_i values are randomly (but uniformly) sampled, this is the Monte-Carlo integration technique. As random generators are not perfect, we rather talk about *quasi*-Monte-Carlo technique. We encourage the reader to consult James [Jam80] for a detailed review on the Monte-Carlo method.

3.2. The neutron weight

A totally realistic semi-classical simulation will require that each neutron is at any time either present or lost. In many instruments, only a very small fraction of the initial neutrons will ever be detected, and simulations of this kind will therefore waste much time in dealing with neutrons that never hit the relevant detector or monitor.

An important way of speeding up calculations is to introduce a neutron "weight factor" for each simulated neutron ray and to adjust this weight according to the path of the ray. If *e.g.* the reflectivity of a certain optical component is 10%, and only reflected neutrons ray are considered later in the simulations, the neutron weight will be multiplied by 0.10 when passing this component, but every neutron is allowed to reflect in the component. In contrast, the totally realistic simulation of the component would require in average ten incoming neutrons for each reflected one.

Let the initial neutron weight be p_0 and let us denote the weight multiplication factor in the j'th component by π_j. The resulting weight factor for the neutron ray after passage of the n components in the instrument becomes the product of all contributions

$$p = p_n = p_0 \prod_{j=1}^{n} \pi_j. \tag{3.2}$$

Each adjustment factor should be $0 < \pi_j < 1$, except in special circumstances, so that total flux can only decrease through the simulation, see section 3.3. For convenience, the value of p is updated (within each component) during the simulation.

Simulation by weight adjustment is performed whenever possible. This includes

- Transmission through filters and windows.
- Transmission through Soller blade collimators and velocity selectors (in the approximation which does not take each blade into account).
- Reflection from monochromator (and analyzer) crystals with finite reflectivity and mosaicity.
- Reflection from guide walls.
- Passage of a continuous beam through a chopper.
- Scattering from all types of samples.

3.2.1. Statistical errors of non-integer counts

In a typical simulation, the result will consist of a count of neutrons histories ("rays") with different weights. The sum of these weights is an estimate of the mean number of neutrons hitting the monitor (or detector) per second in a “real” experiment. One may write the counting result as

$$I = \sum_i p_i = N\overline{p}, \tag{3.3}$$

where N is the number of rays hitting the detector and the horizontal bar denotes averaging. By performing the weight transformations, the (statistical) mean value of I is unchanged. However, N will in general be enhanced, and this will improve the accuracy of the simulation.

To give an estimate of the statistical error, we proceed as follows: Let us first for simplicity assume that all the counted neutron weights are almost equal, $p_i \approx \overline{p}$, and that we observe a large number of neutrons, $N \geq 10$. Then N almost follows a normal distribution with the uncertainty $\sigma(N) = \sqrt{N}$ [1]. Hence, the statistical uncertainty of the observed intensity becomes

$$\sigma(I) = \sqrt{N}\overline{p} = I/\sqrt{N}, \tag{3.4}$$

as is used in real neutron experiments (where $\overline{p} \equiv 1$). For a better approximation we return to Eq. (3.3). Allowing variations in both N and $\overline{p}$, we calculate the variance of the resulting intensity, assuming that the two variables are statistically independent:

$$\sigma^2(I) = \sigma^2(N)\overline{p}^2 + N^2\sigma^2(\overline{p}). \tag{3.5}$$

Assuming as before that N follows a normal distribution, we reach $\sigma^2(N)\overline{p}^2 = N\overline{p}^2$. Further, assuming that the individual weights, p_i, follow a Gaussian distribution (which in some cases is far from the truth) we have $N^2\sigma^2(\overline{p}) = \sigma^2(\sum_i p_i) = N\sigma^2(p_i)$ and reach

$$\sigma^2(I) = N\left(\overline{p}^2 + \sigma^2(p_i)\right). \tag{3.6}$$

The statistical variance of the p_i's is estimated by $\sigma^2(p_i) \approx (\sum_i p_i^2 - N\overline{p}^2)/(N-1)$. The resulting variance then reads

$$\sigma^2(I) = \frac{N}{N-1}\left(\sum_i p_i^2 - \overline{p}^2\right). \tag{3.7}$$

For almost any positive value of N, this is very well approximated by the simple expression

$$\sigma^2(I) \approx \sum_i p_i^2. \tag{3.8}$$

As a consistency check, we note that for all p_i equal, this reduces to eq. (3.4)

In order to compute the intensities and uncertainties, the monitor/detector components in McStas will keep track of $N = \sum_i p_i^0$, $I = \sum_i p_i^1$, and $M_2 = \sum_i p_i^2$.

3.3. Weight factor transformations during a Monte Carlo choice

When a Monte Carlo choice must be performed, *e.g.* when the initial energy and direction of the neutron ray is decided at the source, it is important to adjust the neutron weight

[1] This is not correct in a situation where the detector counts a large fraction of the neutron rays in the simulation, but we will neglect that for now.

so that the combined effect of neutron weight change and Monte Carlo probability of making this particular choice equals the actual physical properties we like to model.

Let us follow up on the simple example of transmission. The probability of transmitting the real neutron is P, but we make the Monte Carlo choice of transmitting the neutron ray each time: $f_{\mathrm{MC}} = 1$. This must be reflected on the choice of weight multiplier $\pi_j = P$. Of course, one could simulate without weight factor transformation, in our notation written as $f_{\mathrm{MC}} = P, \pi_j = 1$. To generalize, weight factor transformations are given by the master equation

$$f_{\mathrm{MC}} \pi_j = P. \tag{3.9}$$

This probability rule is general, and holds also if, e.g., it is decided to transmit only half of the rays ($f_{\mathrm{MC}} = 0.5$). An important different example is elastic scattering from a powder sample, where the Monte-Carlo choices are the particular powder line to scatter from, the scattering position within the sample and the final neutron direction within the Debye-Scherrer cone. This weight transformation is much more complex than described above, but still boils down to obeying the master transformation rule 3.9.

3.3.1. Direction focusing

An important application of weight transformation is direction focusing. Assume that the sample scatters the neutron rays in many directions. In general, only neutron rays in some of these directions will stand any chance of being detected. These directions we call the *interesting directions*. The idea in focusing is to avoid wasting computation time on neutrons scattered in the other directions. This trick is an instance of what in Monte Carlo terminology is known as *importance sampling*.

If *e.g.* a sample scatters isotropically over the whole 4π solid angle, and all interesting directions are known to be contained within a certain solid angle interval $\mathbf{\Delta\Omega}$, only these solid angles are used for the Monte Carlo choice of scattering direction. This implies $f_{\mathrm{MC}}(\Delta\Omega) = 1$. However, if the physical events are distributed uniformly over the unit sphere, we would have $P(\Delta\Omega) = \Delta\Omega/(4\pi)$, according to Eq. (3.9). One thus ensures that the mean simulated intensity is unchanged during a "correct" direction focusing, while a too narrow focusing will result in a lower (*i.e.* wrong) intensity, since we cut neutrons rays that should have reached the final detector.

3.4. Adaptive and Stratified sampling

Another strategy to improve sampling in simulations is *adaptive importance sampling* (also called variance reduction technique), where McStas during the simulations will determine the most interesting directions and gradually change the focusing according to that. Implementation of this idea is found in the **Source_adapt** and **Source_Optimizer** components.

An other class of efficiency improvement technique is the so-called *stratified sampling*. It consists in partitioning the event distributions in representative sub-spaces, which are then all sampled individually. The advantage is that we are then sure that each sub-space

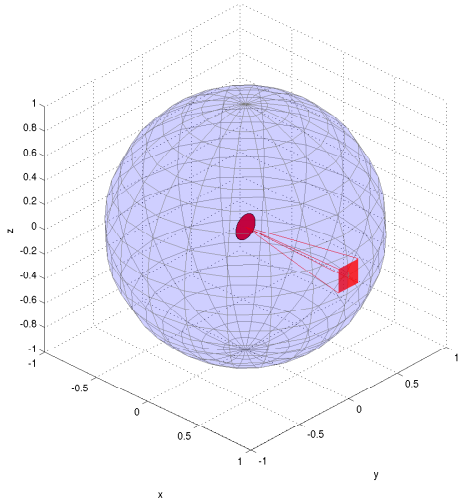


Figure 3.1.: Illustration of the effect of direction focusing in McStas. Weights of neutrons emitted into a certain solid angle are scaled down by the full unit sphere area.

is well represented in the final integrals. This means that instead of shooting N events, we define D partitions and shoot $r = N/D$ events in each partition. In conjunction with adaptive sampling, we may define partitions so that they represent 'interesting' distributions, e.g. from events scattered on a monochromator or a sample. The sum of partitions should equal the total space integrated by the Monte Carlo method, and each partition must be sampled randomly.

In the case of McStas, an ad-hoc implementation of adaptive stratified is used when repeating events, such as in the Virtual sources (Virtual_input, Vitess_input, Virtual_mcnp_input, Virtual_tripoli4_input) and when using the SPLIT keyword in the TRACE section on instrument descriptions. We emphasize here that the number of repetitions r should not exceed the dimensionality of the Monte Carlo integration space (which is $d = 10$ for neutron events) and the dimensionality of the partition spaces, i.e. the number of random generators following the stratified sampling location in the instrument.

3.5. Accuracy of Monte Carlo simulations

When running a Monte Carlo, the meaningful quantities are obtained by integrating random events into a single value (e.g. flux), or onto an histogram grid. The theory [Jam80] shows that the accuracy of these estimates is a function of the space dimension d and the number of events N. For large numbers N, the central limit theorem provides an estimate of the relative error as $1/\sqrt{N}$. However, the exact expression depends on the random distributions.

Records	Accuracy
10^3	10 %
10^4	2.5 %
10^5	1 %
10^6	0.25 %
10^7	0.05 %

Table 3.1.: Accuracy estimate as a function of the number of statistical events used to estimate an integral with McStas.

McStas uses a space with $d = 10$ parameters to describe neutrons (position, velocity, spin, time). We show in Table 3.1 a rough estimate of the accuracy on integrals as a function of the number of records reaching the integration point. This stands both for integrated flux, as well as for histogram bins - for which the number of events per bin should be used for N.

4. Running McStas

This chapter describes usage of the McStas simulation package. In case of problems regarding installation or usage, the McStas mailing list [Mcsa] or the authors should be contacted.

Performing a simulation using McStas can be divided into the following steps/elements

- The structure of McStas is illustrated in Figure 4.1.
- To use McStas, an instrument definition file describing the instrument to be simulated must be written. Alternatively, an example instrument file can be obtained from the `examples/` directory in the distribution or from another source.
- The input files (instrument and component files) are written in the McStas meta-language and are edited either by using your favorite editor or by using the built-in editor of the graphical user interface (`mcgui`).
- Next, the McStas compiler `mcstas` is invoked to translate the instrument and component files into a C program. The program `mcstas` itself is written in C, using the parser *flex* and the compiler compiler *bison*.
- The resulting C program can then be compiled with a C compiler and run in combination with various front-end programs for example to present the intensity at the detector as a motor position is varied.
- The output data may be analyzed and visualized in the same way as regular experiments by using the data handling and visualization tools in McStas. As of McStas 3.x, these front-ends (`mcplot`, `mcdisplay`, `mcresplot`, ...) are all implemented in Python, using `matplotlib`, `pyqtgraph` and browser/HTML (D3.js) back-ends, see section 4.5. Further data output formats including NeXus are available, see section 4.6.

4.1. Installation and updates

For installation notes, see the web site [Mcsa]. In case of problems, write the support mailing list, or contact the authors.

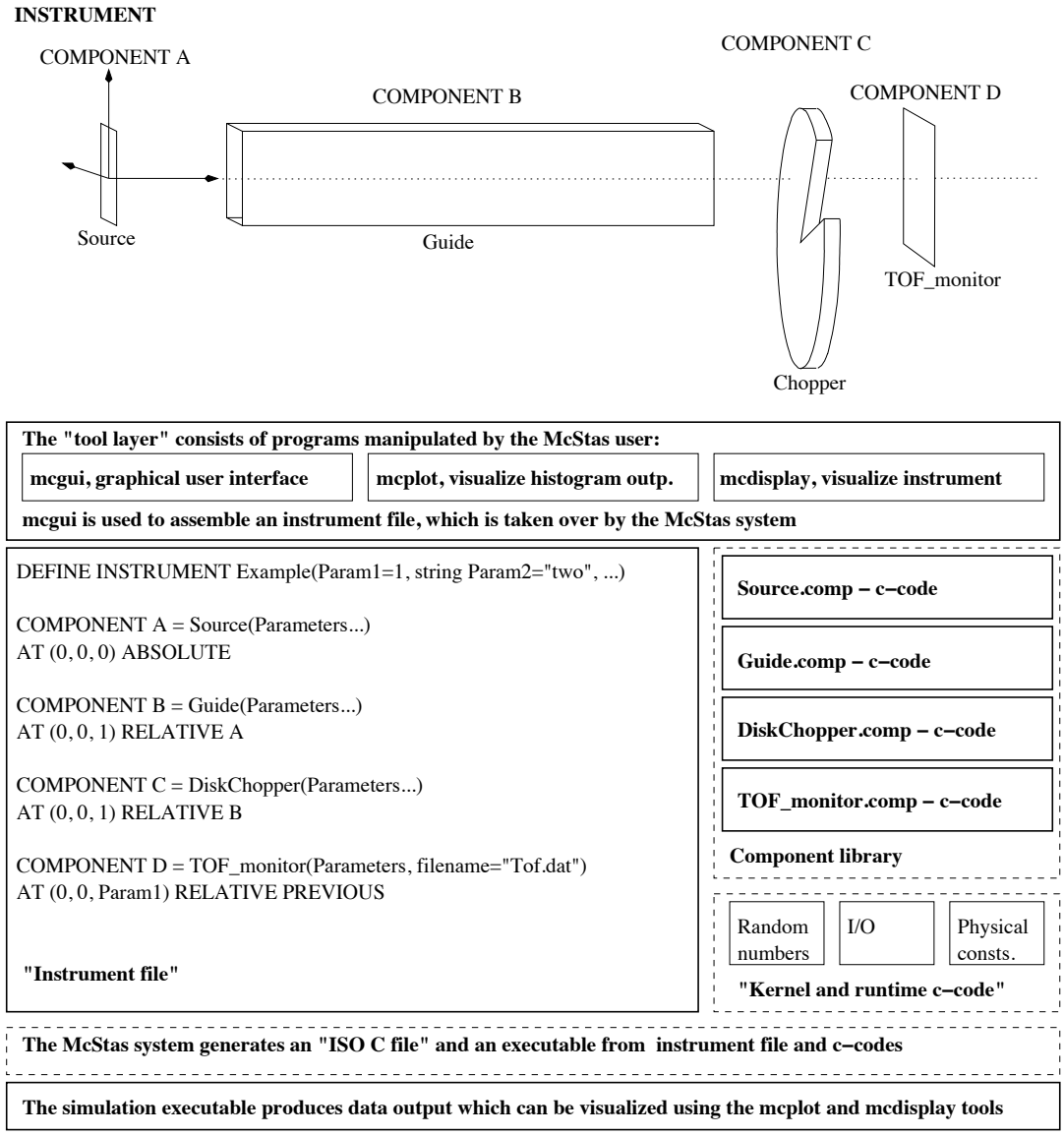


Figure 4.1.: An illustration of the structure of McStas.

Installation via conda-forge (recommended)

The recommended installation method for all supported platforms—Linux, macOS, and Windows—is via the `conda-forge` channel. Once a conda-compatible environment manager such as Miniforge or Mambaforge is available, McStas can be installed into a fresh conda environment:

```
conda create -n mcstas -c conda-forge mcstas
conda activate mcstas
mcgui
```

On Windows a dedicated `mcstas-conda` batch script is provided by the installer. After it completes, use the *mcstas-shell* desktop shortcut and issue `mcgui` to start the graphical interface. Arm64 Windows is also supported via this route.

For full platform-specific installation instructions, including Linux package management and macOS options, see:

`https://github.com/mccode-dev/McCode/tree/main/INSTALL-McStas`

4.1.1. Important note for Windows users

It is a known problem that some of the McStas tools do not support filenames / directories with spaces. We recommend avoiding spaces in the paths used for McStas work directories and instrument files. As of McStas 3.x, all tools (`mcgui`, `mcrun`, `mcplot`, `mcdisplay`, ...) are pure Python and are bundled with the `conda-forge` package, so no separate Perl installation is required on any platform, including Windows.

4.1.2. New releases of McStas

Releases of new versions of a software package can today be carried out more or less continuously. However, users do not update their software on a daily basis, and as a compromise we have adopted the following policy of McStas.

- The versions 3.5.x will possibly contain bug fixes and minor new functionality. A new manual will, however, not be released and the modifications are documented on the McStas web-page. The extensions of the forthcoming version 3.5.x are also listed on the web, and new versions may be released quite frequently when it is requested by the user community.

4.2. Brief introduction to the graphical user interface

This section gives an ultra-brief overview of how to use McStas once it has been properly installed. It is intended for those who do not read manuals if they can avoid it. For details on the different steps, see the following sections. This section uses the `BNL_H8.instr` file supplied in the `examples/BNL/BNL_H8` directory of the McStas distribution.

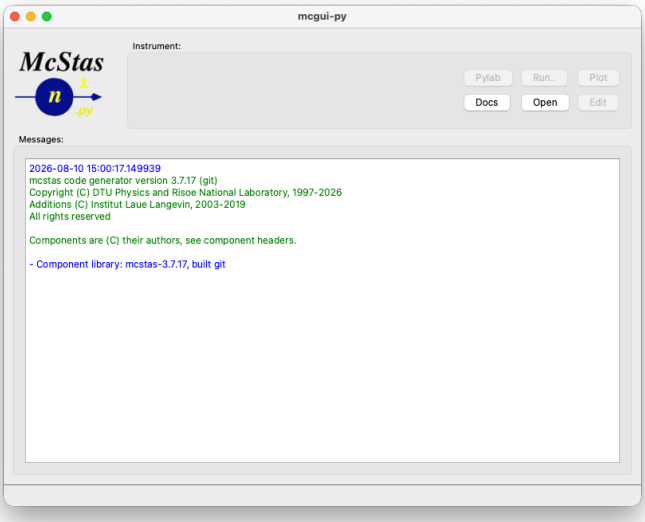


Figure 4.2.: The graphical user interface `mcgui`.

To start the graphical user interface of McStas, run the `mcgui` command which will open a window with a number of menus, see figure 4.2. To load an instrument, select "New from Template" (formerly known as the "Neutron site" menu) and open the template/example `BNL/BNL_H8`. Next, check that the current plotting backend setting ("Preferences" in the **File** menu, or the `--autoplotter` option of `mcrun`) corresponds to your system setup:

- by editing the user configuration file, see `mcrun --edit-user-config`,

Next, select "Run simulation" from the "Simulation" menu. The McStas compiler `mcstas` will translate the definition into an executable program. Then `mcgui` will pop up a dialog window. Type a value for the "ROT" parameter (*e.g.* 90), check the "Plot results" option, and select "Start". The simulation will run, and when it finishes after a while the results will be plotted in a window, using one of the Python plotting back-ends (`matplotlib`, `pyqtgraph` or an HTML/browser view), see section 4.5.6. For other options, execute `mcplot --help` to get help.

To visualize or debug the simulation graphically, repeat the steps but check the "Trace" option instead of the "Simulate" option. A window will pop up showing a sketch of the instrument, drawn by the `mcdisplay` front-end (section 4.5.5), again using one of the `matplotlib`, `pyqtgraph` or webGL back-ends.

For a slightly longer gentle introduction to McStas, see the McStas tutorial (available from [Mcsa]), and as of version 3.5 built into the `mcgui` help menu. For more technical details, read on from section 4.3

4.3. Running the instrument compiler

This section describes how to run the McStas compiler `mcstas` manually. Often, it will be more convenient to use the front-end program `mcgui` (section 4.5.1) or `mcrun` (section 4.5.2), which run the compilation and the simulations automatically.

Upon a command of the form

```
mcstas name.instr
```

the compiler `mcstas` will read the instrument definition `name.instr`, written in the McStas meta-language, and translate it into a Monte Carlo simulation program in the programming language C. The output is by default written to a file in the current directory with the same name as the instrument file, but with extension `.c` rather than `.instr`. This can be overridden using the `-o` option as follows:

```
mcstas -o code.c name.instr
```

which gives the output in the file `code.c`. A single dash '`-`' may be used for both input and output filename to represent standard input and standard output, respectively.

4.3.1. Code generation options

By default, the code generated by `mcstas` is ISO-C with some extensions (currently the only extension is the creation of new directories, which is not possible in pure ISO-C). The use of extensions may be disabled with the `-p` or `--portable` option. With this option, the output is strictly ISO-C compliant, at the cost of some slight reduction in capabilities.

By default (`-t` or `--trace`) the generated code also contains special "trace" code, which makes it possible to get a complete trace of the path of every neutron ray through the instrument, as well as the position and orientation of every component. This is mainly used by the `mcdisplay` front-end as described in section 4.5.5; the `--no-trace` option leaves it out.

The code generation options can also be controlled by using preprocessor macros in the C compiler, without the need to re-run `mcstas`. If the preprocessor macro `MC_PORTABLE` is defined, the same result is obtained as with the `--portable` option. The effect of the `--trace` option may be obtained by defining the `MC_TRACE_ENABLED` macro. Most Unix-like C compilers allow preprocessor macros to be defined using the `-D` option, e.g.

```
cc -DMC_TRACE_ENABLED -DMC_PORTABLE ...
```

Finally, the `--verbose` option will list the components and libraries being included in the instrument. The `--source` option embeds a copy of the instrument source code in the generated executable, and `-v`/`--version` (or `--version-num`) prints the full version header (or just the version number).

Alternative code generator: `mccode-antlr`

From McStas 3.5 onwards an alternative code generator, `mcstas-antlr`, is available alongside the classic generator. It is built on the ANTLR parser framework rather than the traditional `lex`/`yacc` toolchain, is implemented primarily in Python, and is a candidate replacement for the default generator in future releases.

Select it on a per-run basis with the `-cogen` flag:

```
mcrun --cogen=mcstas-antlr MyInstrument.instr -n 1e8
```

The active code generator can also be set permanently by editing the `MCCOGEN` field in `mccode_config.json`. This file is accessible from the command line with:

```
mcrun --edit-user-config
```

or via the *Save/Edit configuration* dialogue inside `mcgui`.

At the time of the McStas 3.6 release, `mcstas-antlr` is close to feature-complete for CPU simulations with a few more issues for GPU/OpenACC simulations.

Alternative code generator: `mcstas-pygen`

A third code generator, `mcstas-pygen`, targets McStasScript [Mcsb] (a Python package for building and running McStas instruments) rather than C: given an `.instr` file, it generates an equivalent McStasScript Python module instead of a C program. Besides `-o FILE` (output file) it accepts `--instrument-name=NAME` (name of the generated McStasScript instrument; by default the instrument name followed by `_generated`), `--no-tests` (do not translate `%Test` (or `%Example`) lines into McStasScript tests), `--lint` (generate a script for McStasScript-style diagnostic linting), `--verbose` and the version options described above. The C-generator-only options (`-I`, `-t`, `-p`, `--no-main`, `--no-runtime`) are rejected. The generated module exposes a `make()` function returning a McStasScript instrument object:

```
import mcstasscript as ms
from INSTRUMENT_generated import make
instr = make()
instr.show_diagram()
instr.show_instrument()
data = instr.backengine()
ms.make_sub_plot(data)
```

The generator is normally invoked indirectly, via `mcstas-jupylab` (section 4.5.3), rather than run directly by the user.

4.3.2. Specifying the location of files

The McStas compiler `mcstas` needs to be able to find various files during compilation, some explicitly requested by the user (such as component definitions and files referenced by `%include`), and some used internally to generate the simulation executable. McStas

looks for these files in three places: first in the current directory, then in a list of directories given by the user, and finally in a special McStas directory. Usually, the user will not need to worry about this as `mcstas` will automatically find the required files. But if users build their own component library in a separate directory or if `mcstas` is installed in an unusual way, it will be necessary to tell the compiler where to look for the files.

The location of the special McStas directory is set when `mcstas` is compiled. It defaults to `/usr/share/mcstas/version` on Debian and derivatives, to `/usr/local/mcstas/version` on RedHat and derivatives and on other Unix-like systems, including Mac OS X, where it is a link to the actual location `/Applications/McStas-version.app/Contents/Resources/mcstas/version`, and `C:\mcstas-version\lib` on Windows systems, but it can be changed to something else, see the installation instructions for details.

The location can be overridden by setting the environment variable `MCSTAS`:

```
1    setenv MCSTAS /home/joe/mcstas
```

for csh/tcsh users, or

```
1    export MCSTAS=/home/joe/mcstas
```

for bash/Bourne shell users. Windows users should define `MCSTAS` from the menu 'Start/Settings/Control Panel/System/Advanced/Environment Variables' by creating `MCSTAS` with the value `C:\mcstas\lib`

To make `mcstas` search additional directories for component definitions and include files, use the `-I` switch:

```
1    mcstas -I/home/joe/components -I/home/joe/neutron/include name.instr
```

Multiple `-I` options can be given, as shown.

4.3.3. Embedding the generated simulations in other programs

By default, `mcstas` will generate a stand-alone C program, which is what is needed in most cases. However, for advanced usage, such as embedding the generated simulation in another program or even including two or more simulations in the same program, a stand-alone program is not appropriate. For such usage, `mcstas` provides the following options:

- `--no-main` This option makes `mcstas` omit the `main()` function in the generated simulation program. The user must then arrange for the function `mcstas_main()` to be called in some way.
- `--no-runtime` Normally, the generated simulation program contains all the run-time C code necessary for declaring functions, variables, etc. used during the simulation. This option makes `mcstas` omit the run-time code from the generated simulation program, and the user must then explicitly link with the file `mcstas-r.c` as well as other shared libraries from the McStas distribution.

Users that need these options are encouraged to contact the authors for further help.

4.3.4. Running the C compiler

After the source code for the simulation program has been generated with `mcstas`, it must be compiled with the C compiler to produce an executable. Since the generated C code obeys the ISO-C standard, it should be easy to compile it using any ISO-C (or C++) compiler. *E.g.* a typical Unix-style command would be

```
1    cc -O -o name.out name.c -lm
```

The McStas team recommends these compiler alternatives for the Intel (and AMD) hardware architectures:

A `gcc` which is a very portable, open source, ISO-C compatible c compiler, available for most platforms. For Linux it is usually part of your distribution, for Windows the McStas distribution package includes a version of `gcc` (in the Dev-CPP subpackage), and for Mac OS X `gcc` is part of the Xcode tools package available on the installation medium.

B `icc` or the Intel c compiler is available for Linux, Mac OS and Windows systems and is a commercial software product. Generally, simulations run with the Intel compiler are **a factor of 2 faster** than the identical simulation run using `gcc`. To use `icc` with McStas on Linux or Mac OS X, set the environment variables

- `MCSTAS_CC=icc`
- `MCSTAS_CFLAGS="-g -O2 -wd177,266,1011,181"`

To use `icc` with MPI on Unix system (see Section 4.7) installations, it seems that *editing* the mpicc shell script and setting the CC variable to `"icc"` is the only requirement! On Windows, the Intel c compiler is 'icl', not 'icc' and has a dependency for Microsoft Visual C++. If you have both these softwares available, running McStas with the Intel compiler should be possible (currently untested by the McStas developer team).

The `-O` option typically enables the optimization phase of the compiler, which can make quite a difference in speed of `mcstas`-generated simulations. The `-o name.out` sets the name of the generated executable. The `-lm` options is needed on many systems to link in the math runtime library (like the cos() and sin() functions).

Monte Carlo simulations are computationally intensive, and it is often desirable to have them run as fast as possible. Some success can be obtained by adjusting the compiler optimization options. Here are some example platform and compiler combinations that have been found to perform well (up-to-date information will be available on the McStas WWW home page [Mcsa]):

- Intel x86 ("PC") with Linux and GCC, using options `gcc -O3`.
- Intel x86 with Linux and EGCS (GCC derivate) using options `egcc -O6`.
- Intel x86 with Linux and PGCC (pentium-optimized GCC derivate), using options `gcc -O6 -mstack-align-double`.

- HPPA machines running HPUX with the optional ISO-C compiler, using the options `-Aa +Oall -Wl,-a,archive` (the `-Aa` option is necessary to enable the ISO-C standard).
- SGI machines running Irix with the options `-Ofast -o32 -w`

Optimization flags will typically result in a speed improvement by a factor about 3, but the compilation of the instrument may be 5 times slower.

A warning is in place here: it is tempting to spend far more time fiddling with compiler options and benchmarking than is actually saved in computation times. Even worse, compiler optimizations are notoriously buggy; the options given above for PGCC on Linux and the ISO-C compiler for HPUX have been known to generate *incorrect code* in some compiler versions. `mcstas` actually puts an effort into making the task of the C compiler easier, by in-lining code and using variables in an efficient way. As a result, McStas simulations generally run quite fast, often fast enough that further optimizations are not worthwhile. Also, optimizations are highly time and memory consuming during compilation, and thus may fail when dealing with large instrument descriptions (e.g. more that 100 elements). The compilation process is simplified when using components of the library making use of shared libraries (see `SHARE` keyword in chapter 5). Refer to section 4.4.4 for other optimization methods.

4.4. Running the simulations

Once the simulation program has been generated by the McStas compiler `mcstas` and an executable has been obtained with the C compiler, the simulation can be run in various ways.

Simple McStas options

In this section, the most common simulation parameters are discussed. For a full list, please consult Tables 4.1 and 4.2.

The simplest way is to run it directly from the command line or shell:

```
1 ./name.out
```

Note the leading ".", which is needed if the current directory is not in the path searched by the shell. When used in this way, the simulation will prompt for the values of any instrument parameters such as angular settings, and then run the simulation. Default instrument parameter values (see section 5.3), if any, will be indicated and entered when hitting the `Return` key. This way of running McStas will only give data for one instrument setting which is normally sufficient for *e.g.*, time-of-flight, SANS or powder instruments, but not for *e.g.* continuous-beam reflectometers or triple-axis spectrometers where a scan over various instrument settings is required. Often the simulation will be run using one of several available front-ends, as described in the next section. These front-ends help manage output from the potentially many detectors in the instruments, as well as running the simulation for each data point in a scan.

The generated simulations accept a number of options and arguments. The full list can be obtained using the `--help` option:

```
./name.out --help
```

The values of instrument parameters may be specified as arguments using the syntax *name=val*. For example

```
./BNL_H8.out lambda=2.36
```

The number of neutron histories to simulate may be set using the `--ncount` or `-n` option, for example `--ncount=2e5`. The initial seed for the random number generator is by default chosen based on the current time so that it is different for each run. However, for debugging purposes it is sometimes convenient to use the same seed for several runs, so that the same sequence of random numbers is used each time. To achieve this, the random seed may be set using the `--seed` or `-s` option.

By default, McStas simulations write their results into several data files in the current directory, overwriting any previous files stored there. The `--dir=`*dir* or `-d`*dir* option causes the files to be placed instead in a newly created directory *dir* (to prevent overwriting previous results an error message is given if the directory already exists).

The complete list of options and arguments accepted by McStas simulations appears in Tables 4.1 and 4.2.

4.4.1. Choosing an output data file format

Data files contain header lines with information about the simulation from which they originate. Use `--format` to select among the available formats (see below).

The format of the output files from McStas simulations is described in more detail in section 4.6. It may be chosen either with `--format=FORMAT` for each simulation or globally by setting the MCSTAS_FORMAT environment variable. The available format list is obtained using the `name.out --help` option. McStas can presently generate data in the McStas/McCode text format or the NeXus format (see section 4.6.2).

It is also possible to read and write neutron event lists using the `MCPL_input`/`MCPL_output` components (see the Component Manual), which use the portable, compressed, multi-code MCPL event format rather than any particular code's native event format. This is the recommended way to exchange neutron events between McStas/McXtrace and other Monte Carlo codes. The older, code-specific event-file components (`Virtual_input`/`Virtual_output`, `Virtual_mcnp_input`/`Virtual_mcnp_output`, `Virtual_tripoli4_input`/`Virtual_tripoli4_output` `Vitess_input`/`Vitess_output`) still exist for backward compatibility, but have moved to the `obsolete` component category (see the Component Manual) and are no longer recommended for new instrument work in favour of MCPL.

Additionally, adding the `raw` keyword to the FORMAT will produce raw $[N, p, p^2]$ data sets instead of $[N, p, \sigma]$ (see Section 3.2.1). The former representation is fully additive, and thus enables to add results from separate simulations. Other acceptable format modifiers are `transpose` to transpose data matrices and `append` to concatenate data to existing files.

4.4.2. Basic import and plot of results

The previous example will result in a `mccode.sim` index file plus one data file per monitor in the output directory. These are plain-text McStas/McCode format files (see section 4.6) that can be read directly with NumPy (`numpy.loadtxt`) or any other language capable of reading text tables, or, more conveniently, with the `mcplot` front-end and its underlying data loader (`mccodelib.mcplotloader`), see section 4.5.6.

When using the HTML plotting back-end (`mcplot-html`), simulation results are rendered as an interactive, self-contained web page (using D3.js) rather than being saved as static images.

The full, up-to-date list of options accepted by `mcrun` and by the generated `<instr>.out` executable is given in Tables 4.1 and 4.2 below. These tables are generated automatically from the `mcrun` Python source (`tools/Python/mcrun/mcrun.py`) by the script `gen_mcrun_options_table.py` in this directory, so that they cannot silently go out of date; run `mcrun --help` for the same information at the command line.

Table 4.1.: `mcrun`-specific options (compilation, scanning, optimisation, MPI/OpenACC).

Option	**Description**
`-c` `-force-compile`	force rebuilding of instrument
`-cogen` *cogen*	Choice of code-generator (implies -c)
`-C` `-c-lint`	Use c-linter (e.g. cppcheck) to lint the generated code. Configure linter via mccode_config.json. Implies -c and -v, but also NO simulation will be run.
`-I`	Append to McCode search path (implies -c)
`-D1` *D1*	Set extra -D args (implies -c)
`-D2` *D2*	Set extra -D args (implies -c)
`-D3` *D3*	Set extra -D args (implies -c)
`-p` `-param` *FILE*	Read parameters from file FILE
`-N` `-numpoints` *NP*	Set number of scan points
`-seeds` *SEEDS*	Set range of seeds to scan (each must be: SEED != 0)
`-L` `-list`	Use a fixed list of points for linear scanning
`-M` `-multi`	Run a multi-dimensional scan
`-scan_split` *scan_split*	Scan by parallelising steps as individual cpu threads. Initialise by number of wanted threads (e.g. your number of cores).
`-autoplot`	Open plotter on generated dataset

Option	Description
`-invcanvas`	Forward request for inverted canvas to plotter
`-autoplotter`	Specify the plotter used with –autoplot
`-embed`	Store copy of instrument file in output directory *(default: True)*
`-mpi` *NB_CPU*	Spread simulation over NB_CPU machines using MPI
`-openacc`	parallelize using openacc
`-funnel`	funneling simulation flow, e.g. for mixed CPU/GPU
`-machines` *machines*	Defines path of MPI machinefile to use in parallel mode
`-optimise-file` *FILE*	Store scan results in FILE (defaults to: "mccode.dat")
`-no-cflags`	Disable optimising compiler flags for faster compilation
`-no-main`	Do not generate a main(), e.g. for use with mcstas2vitess.pl. Implies -c
`-verbose`	Enable verbose output
`-write-user-config`	Generate a user config file
`-edit-user-config`	Generate and edit user config file in EDITOR
`-override-config` *PATH*	Load config file from specific dir
`-optimize`	Optimize instrument variable parameters to maximize monitors
`-optimize-maxiter` *optimize_maxiter*	Maximum number of optimization iterations to perform. Default=1000 *(default: 1000)*
`-optimize-tol` *optimize_tol*	Tolerance for optimization termination. When optimize-tol is specified, the selected optimization algorithm sets some relevant solver-specific tolerance(s) equal to optimize-tol
`-optimize-method` *optimize_method*	Optimization solver in [...] (default: [...]) You can use your custom method method(fun, x0, args, **kwargs, **options). Please refer to scipy documentation for proper use of it: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimiz
`-optimize-eval` *optimize_eval*	Optimization expression to evaluate for each detector "d" structure. You may combine: "d.intensity" The detector intensity; "d.error" The detector intensity uncertainty; "d.values" An array with [intensity, error, counts]; "d.X0 d.Y0" Center of signal (1st moment); "d.dX d.dY" Width of signal (2nd moment). Default is "d.intensity". Examples are: "d.intensity/d.dX" and "d.intensity/d.dX/d.dY"

Option	Description
`-optimize-minimize`	Choose to minimize the monitors instead of maximize
`-optimize-monitor` *optimize_monitor*	Name of a single monitor to optimize (default is to use all)
`-showcfg` *ITEM*	Print selected cfg item and exit (paths are resolved and absolute). Allowed values are particle.

Table 4.2.: Instrument/simulation options (also accepted directly by the generated `<instr>.out` executable).

Option	Description
`-s` `-seed` *SEED*	Set random seed (must be: SEED != 0)
`-n` `-ncount` *COUNT*	Set number of particles to simulate *(default: 1000000)*
`-t` `-trace` *trace*	Enable trace of particles through instrument *(default: 0)*
`-no-trace` *notrace*	Disable trace of particles in instrument (combine with -c)
`-y` `-yes`	Assume any default parameter value in instrument
`-d` `-dir` *DIR*	Put all data files in directory DIR. If unspecified INSTRUMENT_TIMESTAMP is used
`-dirprefix` *dirprefix*	Put all data files in directory PREFIX_TIMESTAMP
`-dirsuffix` *dirsuffix*	Put all data files in directory INSTRUMENT_DIRSUFFIX
`-a` `-append`	Append data files to those already in directory DIR
`-format` *FORMAT*	Output data files using format FORMAT, usually McCode or NeXus (format list obtained from <instr>.out -h) *(default: McCode)*
`-bufsiz` *BUFSIZ*	Monitor_nD list/buffer-size (defaults to [...])
`-vecsize` *VECSIZE*	vector length in OpenACC parallel scenarios
`-numgangs` *NUMGANGS*	number of 'gangs' in OpenACC parallel scenarios
`-gpu_innerloop` *INNERLOOP*	Maximum particles in an OpenACC kernel run. (If INNERLOOP is smaller than ncount we repeat)
`-no-output-files`	Do not write any data files
`-i` `-info`	Detailed instrument information
`-list-parameters`	Print the instrument parameters to standard out

Option	Description
-meta-list	Print all metadata defining component names
-meta-defined	Print metadata names for component, or indicate if component:name exists
-meta-type	Print metadata type for component:name
-meta-data	Print metadata for component:name
-g -gravitation -gravity	Enable gravitation for all trajectories
-IDF	Flag to attempt inclusion of XML-based IDF when –format=NeXus (format list obtained from <instr>.out -h)

4.4.3. Interacting with a running simulation

Once the simulation has started, it is possible, under Unix, Linux and Mac OS X systems, to interact with the on-going simulation. This feature is not available when using MPI parallelization.

McStas attaches a signal handler to the simulation process. In order to send a signal to the process, the process-id *pid* must be known. Users may look at their running processes with the Unix 'ps' command, or alternatively process managers like 'top' and 'gtop'. If a *file.out* simulation obtained from McStas is running, the process status command should output a line resembling

```
1 <user> 13277 7140 99 23:52 pts/2    00:00:13    file.out
```

where `user` is your Unix login. The *pid* is there '13277'.

Once known, it is possible to send one of the signals listed in Table 4.3 using the 'kill' unix command (or the functionalities of your process manager), e.g.

```
1     kill -USR2 13277
```

This will result in a message showing status (here 33 % achieved), as well as the position in the instrument of the current neutron.

```
1 # McStas: [pid 13277] Signal 12 detected SIGUSR2 (Save simulation)
2 # Simulation: file (file.instr)
3 # Breakpoint: MyDetector (Trace) 33.37 % (  333654.0/ 1000000.0)
4 # Date       : Wed May  7 00:00:52 2003
5 # McStas: Saving data and resume simulation (continue)
```

followed by the list of detector outputs (integrated counts and files). Finally, sending a `kill 13277` (which is equivalent to `kill -TERM 13277`) will end the simulation before the initial 'ncount' preset.

A typical usage example would be, for instance, to save data during a simulation, plot or analyze it, and decide to interrupt the simulation earlier if the desired statistics has been achieved. This may be done automatically using the `Progress_bar` component.

`USR1`	Request information (status)
`USR2, HUP`	Request information and performs an intermediate saving of all monitors (status and save). This triggers the execution of all `SAVE` sections (see chapter 5).
`INT, TERM`	Save and exit before end (status)

Table 4.3.: Signals supported by McStas simulations.

Whenever simulation data is generated before end (or the simulation is interrupted), the 'ratio' field of the monitored data will provide the level of achievement of the computation (for instance '3.33e+05/1e+06'). Intensities are then usually to be scaled accordingly by the user.

Additionally, any system error will result in similar messages, giving indication about the occurrence of the error (component and section). Whenever possible, the simulation will *try* to save the data before ending. Most errors appear when using a newly written component, in the `INITIALIZE`, `TRACE` or `FINALLY` sections. Memory errors usually show up when C pointers have not been allocated/unallocated before usage, whereas mathematical errors are found when, for instance, dividing by zero.

4.4.4. Optimizing simulation speed

There are various ways to speed up simulations

- Optimize the compilation of the instrument, as explained in section 4.3.4.
- Execute the simulation in parallel on a computer cluster using MPI, or simply across the cores of a single machine, as explained in section 4.7.
- Divide simulation into parts using a file for saving or generating neutron events. In this way, a guide may be simulated only once, saving the neutron events at the guide exit as a file, which is being read quickly by the second simulation part. Use the `MCPL_output` and `MCPL_input` components for this technique (see the Component Manual).
- Use source optimizers like the components Source_adapt or Source_Optimizer. Such component may sometimes not be very efficient, when no neutron importance sampling can be achieved, or may even sometimes alter the simulation results. Be careful and always check results with a (shorter) non-optimized computation.
- Complex components usually take into account additional small effects in a simulation, but are much longer to execute. Thus, simple components should be preferred whenever possible, at least in the beginning of a simulation project.
- The SPLIT keyword may artificially repeat events reaching specified positions in the instrument. This is *very* efficient, but requires to cast random numbers in the

course of the remaining propagation (e.g. at samples, crystals, ...). See section 5.4.6 for details.

A general comment about optimization is that it should be used cautiously, checking that the results are not significantly affected.

4.4.5. Optimizing instrument parameters

Often, the user may wish to optimize the parameters of a simulation, i.e. the best geometry of a given component, for example the optimal curvature of a monochromator.

The choice of the optimization routine, of the simulation quality value to optimize, the initial parameter guess and the simulation length all have a large influence on the results. The user is advised to be cautious when interpreting the optimization results.

Using iFit for optimization

One of the authors of McStas has developed a very flexible and general data analysis and fitting package called iFit [EF14; Ifi] based on Matlab. Matlab itself is not required, as a stand-alone distributable binary of iFit exists.

iFit contains wrapper functionality for compiling and running McStas simulations as object functions, and allows to select many different optimizers, including swarms and other non-gradient methods. Please see the iFit documentation for more information.

Our experience is that iFit together with McStas can be a more robust optimization solution than the McStas built-in `-optimize` mode described next, in particular for problems with many free parameters.

Using the built-in `-optimize` mode

The McStas package comes with a built-in optimization mode to find instrument parameters that maximize (or minimize) the value of all, or some specified, monitors. As of McStas 3.x this is implemented in `mcrun` using the `scipy.optimize.minimize` family of solvers [Sci], replacing the earlier built-in Simplex/Nelder–Mead implementation. Available methods include `powell` (the default), `nelder-mead`, `cg`, `bfgs`, `newton-cg`, `l-bfgs-b`, `tnc`, `cobyla`, `slsqp`, `trust-constr`, `dogleg`, `trust-ncg`, `trust-exact` and `trust-krylov`; consult the SciPy documentation for the properties of each. As with any non-linear optimizer, results depend on the initial guess and the chosen method, and higher-dimensional problems (many free parameters) are not guaranteed to converge to a meaningful, global optimum.

When using `mcrun` (section 4.5.2), the optimization mode is set by using the `--optimize` flag, together with instrument parameter ranges specified exactly as for a scan, e.g. `param=min,max`. The solver is selected with `--optimize-method=METHOD`, the convergence tolerance with `--optimize-tol=TOL`, and the maximum number of iterations with `--optimize-maxiter=N`. By default all monitors are summed as the figure of merit; a single monitor may be targeted instead with `--optimize-monitor=NAME`, and `--optimize-minimize` inverts the sense of the search (minimize rather than maximize). For full control over

the optimized quantity (e.g. optimizing on peak width rather than intensity), use `--optimize-eval=EXPR` with an expression combining the fields of a detector structure `d` (`d.intensity`, `d.error`, `d.values`, `d.X0`/`d.Y0`, `d.dX`/`d.dY`), e.g. `--optimize-eval="d.intensity/d.dX"`. See Table 4.1 for the complete, up-to-date option list.

Example:

```
mcrun templateTAS.instr --optimize --optimize-monitor=focus_monitor A3
    =10,80
```

From `mcgui` (section 4.5.1), one should choose the 'Optimization' execution mode (instead of the Simulation or Trace mode). Then specify the instrument parameters to optimize by indicating their variation range `param=min,max` (e.g. Lambda=1,4) just like parameter scans. Finally, run the simulation. The optimum set of parameters is then printed at the end of the simulation process. You may ask to maximize only a given monitor (instead of all) by selecting its component name in the Run Dialog.

If you would like to maximize the flux at a given monitor, with some divergence constrains, you should for instance simply add a divergence collimator before the monitor. Alternatively, write a new component that produce the required 'figure-of-merit'.

The optimization search interval constrains the evolution of parameters. It should be chosen carefully. In particular it is safer for it to indeed contain a high signal domain, and be preferably symmetric with respect to that maximum.

Using custom optimization routines

For cases the built-in `--optimize` mode (or iFit) does not cover, the user may still write a custom function/script or program that

- inputs the simulation parameters, which are usually numerical values such as TT in the `ISIS_Prisma2` instrument from the `examples` directory of the package.
- builds a command line from these parameters.
- executes that command, and waits until the end of the computation.
- reads the relevant data from the monitors.
- outputs a simulation quality measurement from this data, usually the integrated counts or some peak width.

For instance, for the `ISIS_Prisma2` instrument we could write a function for Python using `scipy.optimize` (or Matlab, if preferred) in order to study the effects of the TT parameter:

```
import subprocess
from scipy.optimize import minimize_scalar
import numpy as np

def instr_value(TT):
    outdir = f"opt_{TT:.3f}"
```

```
    subprocess.run(["mcrun", "ISIS_Prisma2.instr", "-d", outdir, "-n", "1e5
        ",
                    f"TT={TT}", "PHA=22", "PHA1=-3", "PHA2=-2", "PHA3=-1",
                    "PHA4=0", "PHA5=1", "PHA6=2", "PHA7=3", "TTA=44"],
                        check=True)
    # read back the relevant monitor's plain-text McCode data file directly
    data = np.loadtxt(f"{outdir}/mon9.dat")
    return -data[:, 1].sum()   # negative intensity column sum: we minimize
         below

result = minimize_scalar(instr_value, bounds=(-30, -20), method="bounded")
```

will determine the best value of `TT` in order to maximize the mean detector counts. This is essentially what `mcrun -optimize` already automates for you (section 4.4.5); writing a custom driver like this remains useful mainly when the figure of merit cannot be expressed as a simple `--optimize-eval` expression, or when a non-scipy optimizer (e.g. a Matlab/iFit swarm optimizer) is preferred.

4.5. Using simulation front-ends

McStas includes a number of front-end programs that extend the functionality of the simulations. A front-end program is an interface between the user and the simulations, running the simulations and presenting the output in various ways to the user.

The list of available McStas front-end programs is (see `mcdoc --tools`, or run any tool with `-h` for its specific help):

```
    McStas Tools
       mcstas           Main instrument compiler
       mcstas-pygen     Alternative code generator: McStasScript Python
           model
       mcrun            Instrument build and execution utility
       mcstas-jupylab   Opens a generated instrument model in JupyterLab
       mcgui            Graphical User Interface instrument builder
       mcdoc            Component library documentation generator/viewer
       mcplot           Simulation result viewer (default: pyqtgraph backend
           )
       mcplot-html      Simulation result viewer, in a web browser (D3.js)
       mcplotdiff-html  Difference plot between two simulation results, in a
            browser
       mccoplot-html    Overlay (co-plot) of two simulation results' 1D
           monitors
       mcdisplay        Instrument geometry viewer (default: pyqtgraph
           backend)
       mcresplot        Instrument resolution function viewer (Python/
           matplotlib)
       mctest           Self-test / benchmark of the current McStas
           installation
       mcviewtest       Viewer for mctest results, with mcplotdiff-html
           comparisons
    Each tool also exists in dedicated *-matplotlib, *-pyqtgraph, *-html,
```

```
17      *-webgl or *-webgl-classic variants where applicable, see sections
            below.
18      DOC:        Please visit https://www.mcstas.org
```

4.5.1. The graphical user interface (mcgui)

The front-end `mcgui` provides a graphical user interface that interfaces the various parts of the McStas package. It may be started with the single command

```
1       mcgui
```

The mcgui program may optionally be given the name of the instrument file to use.

Dependencies: As of McStas 3.x, `mcgui` is a pure Python/Qt application (PyQt), bundled with the standard conda-forge McStas package; no separate Perl, Tk, or PGPLOT installation is required.

The menus

When the front-end is started the main window is opened (see figure 4.2). This window displays the output from compiling and running simulations, and contains a few menus and buttons for easy navigation. The main purpose of the front-end is to edit and compile instrument definitions, run the simulations, and visualize the results.

The **File** menu has the following features:

File/Open instrument selects the name of an instrument file to be used.

File/Edit current opens a simple editor window with McStas syntax highlighting for editing the current instrument definition. This function is also available from the **Edit** button to the right of the name of the instrument definition in the main window.

File/Spawn editor This starts the editor defined in the environment variable `VISUAL` or `EDITOR` on the current instrument file. It is also possible to start an external editor manually; in any case `mcgui` will recompile instrument definitions as necessary based on the modification dates of the files on the disk.

File/Compile instrument forces a recompile of the instrument definition, regardless of file dates. This is for example useful to pick up changes in component definitions, which the front-end will not notice automatically. This might also be required when choosing MPI and NeXus options.

File/Save log file saves the text in the window showing output of compilations and simulations into a file.

File/Clear output erases all text in the window showing output of compilations and simulations.

File/Preferences Opens the configuration dialog shown in figure 4.3. Several settings can be chosen here:

- Selection of the desired plotting/display backend (`pyqtgraph`, `matplotlib`, `html`, ...) for `mcplot` and `mcdisplay`.
- Choice of code generator (`mccode` or `mccode-antlr`, see section 4.3.1).
- Choice of editor to use when editing instrument files.
- Automatic quotation of strings when inserting in the built-in editor.
- Possibility to *not* optimize when compiling the generated c-code. This is very handy when setting up an instrument model, which requires regular compilations.

To save the chosen settings for your next McStas run, use Save Configuration in the File menu.

File/Save configuration saves user settings from Configuration options and Run dialogue to disk.

File/Quit exits the graphical user interface front-end.

The **Simulation** menu has the following features:

Simulation/Read old simulation prompts for the name of a file from a previous run of a McStas simulation (usually called `mccode.sim`). The file will be read and any detector data plotted using the `mcplot` front-end. The parameters used in the simulation will also be made the defaults for the next simulation run. This function is also available using the "Read" button to the right of the name of the current simulation data.

Simulation/Run simulation opens the run dialog window, explained further below.

Simulation/Plot results plots (using `mcplot`) the results of the last simulation run or spawns a load dialogue to load a set of results.

The **New from Template** menu (known as "Neutron site" in older McStas releases) contains a list of template and example instruments as found in the McStas library, sorted by category/facility (e.g. `Templates`, `ILL`, `ISIS`, `ESS`, `Union_demos`, ... – see chapter 10 for a curated tour). When selecting one of these, a local copy of the instrument description is transferred to the active directory (so that users have modification rights) and loaded. One may then view its source (Edit) and use it directly for simulations/trace (3D View).

The **Tools** menu gathers minor tools.

Tools/Plot current/other results Plot current simulation results and other results.

Tools/Dataset convert/merge Opens a GUI to merge scattered data sets (e.g. from successive runs or an MPI job) or assemble scan sets.

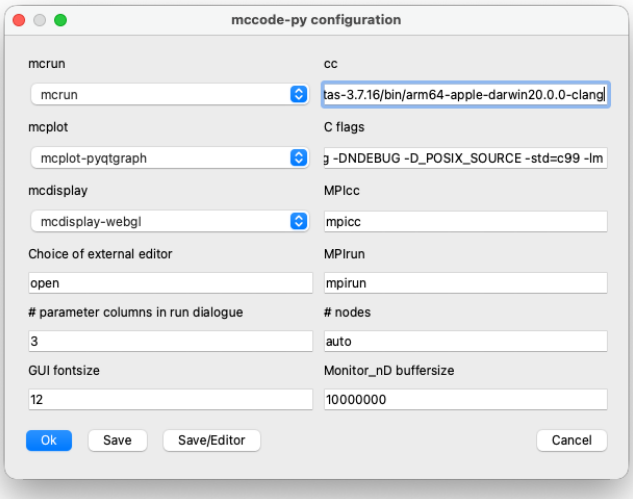


Figure 4.3.: The "configuration options" dialog in `mcgui`.

Tools/Shortcut keys displays the shortcut keys used for running and editing instruments.

The **Help** menu has the following features, through use of `mcdoc` and a web browser. To customize the used web browser, set the `BROWSER` environment variable. If `BROWSER` is not set, `mcgui` uses `netscape/mozilla/firefox` on Unix/Linux and the default browser on Windows.

Help/McStas User manual calls `mcdoc --manual`, brings up the local pdf version of this manual, using a web browser.

Help/McStas Component manual calls `mcdoc --comps`, brings up the local pdf version of the component manual, using a web browser.

Help/Component library index displays the component documentation using the component `index.html` index file.

Help/McStas web page calls `mcdoc --web`, brings up the McStas website in a web browser.

Help/Tutorial opens the McStas tutorial for a quick start.

Help/Current instrument info generates a description web-page of the current edited instrument.

Help/Test McStas installation launches `mctest` to check that the McStas package is installed properly and generates accurate results (see section 4.5).

Help/Generate component index (re-)generates locally the component `index.html`.

The run dialog

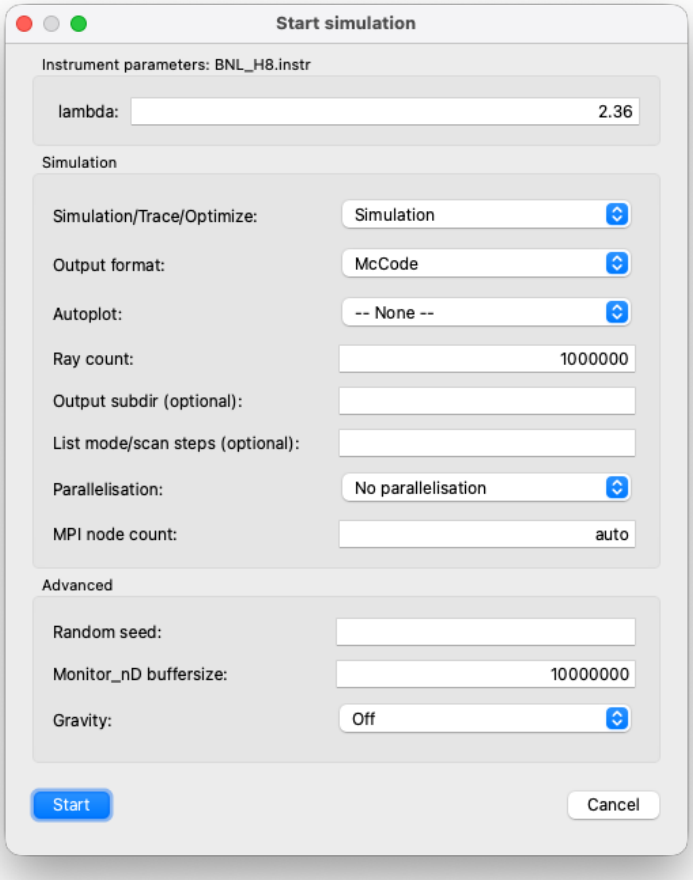


Figure 4.4.: The run dialog in `mcgui`.

The run dialog is used to run simulations. It allows the entry of instrument parameters as well as the specifications of options for running the simulation (see section 4.4 for details). It also allows to run the `mcdisplay` (section 4.5.5) and `mcplot` (section 4.5.6)

front-ends together with the simulation.

The meaning of the different fields is as follows:

Run:Instrument parameters allows the setting of the values for the input parameters of the instrument. The type of each instrument parameter is given in parenthesis after each name. Floating point numbers are denoted by (D) (for the C type "`double`"), (I) denotes integer parameters, and (S) denotes strings. For parameter scans and optimizations, enter the minimum and maximum values to scan/optimize, separated by a comma, e.g. `1,10` and do not forget to set the # **Scanpoints** to more than 1.

Run:Output to allows the entry of a directory for storage of the resulting data files in (like the `--dir` option). If no name is given, the results are stored in the current directory, to be overwritten by the next simulation.

Run:Force Forces McStas to overwrite existing data files

Neutron count sets the number of neutron rays to simulate (the `--ncount` option).

Run:Gravity Activates gravitation handling. Not all components full support the use of gravitation, but all transport in "free space" using the PROP_DT, PROP_Z0 etc. macros will include propagation with gravity. Only local, internal component propagation without the PROP routines will be gravity-less. As a conclusion it is considered safe and to high precision correct to apply the gravitation setting if one takes care to use the Guide_gravity component and other gravity-supporting guide types in combination with non-gravity components that are "small" in size, i.e. samples, lenses, etc.

Run:Random seed/Set seed to selects between using a random seed (different in each simulation) for the random number generator, or using a fixed seed (to reproduce results for debugging).

Run:Simulate/Trace (3D)/Optimize selects between several modes of running the simulation:

- Simulate: perform a normal simulation or a scan when #steps is set to non-zero value
- Trace (3D view): View the instrument in 3D tracing individual neutrons through the instrument
- Optimize: find the optimum value of the simulation parameters in the given ranges (see section 4.4.5).
- Backgrounding (bg): Simulate or Optimize in the background.

Run:# steps / # optim sets the number of simulation to run when performing a parameter scan or the number of iterations to perform in optimization mode.

Run:Plot results – if checked, the `mcplot` front-end will be run after the simulation has finished, and the plot dialog will appear (see below).

Run:Format quick selection of output format (`McCode` or `NeXus`).

Run:Clustering method selects between running locally or via MPI. See section 4.7 on parallel computing for more informations.

Run:Number of nodes sets the number of MPI nodes to use.

Run:Inspect component (Trace mode) will trace only neutron trajectories that reach a given component (e.g. sample or detector).

Run:First component (Trace mode) seletcs the first component to plot (default is first) in order to define a region of interest.

Run:Last component (Trace mode) seletcs the last component to plot (default is first) in order to define a region of interest.

Run:Maximize monitor (Optimization mode) seletcs up to three monitors which integral value should be maximized, varying instrument parameters. If no monitor is selected, the sum of all monitors is optimized.

Run:Start runs the simulation.

Run:Cancel aborts the dialog.

Most of the settings on the run dialog can be saved for your next McStas run using 'Save configuration' in the File menu.

Before running the simulation, the instrument definition is automatically compiled if it is newer than the generated C file (or if the C file is newer than the executable). The executable is assumed to have a `.out` suffix in the filename. NB: If components are changed, automatic compilation is *not* performed. Instead, use the File/Compile menu item in mcgui.

The editor window

The editor window provides a simple editor for creating and modifying instrument definitions. Apart from the usual editor functions, the "Insert" menu provides some functions that aid in the construction of the instrument definitions:

Editor Insert/Instrument template inserts the text for a simple instrument skeleton in the editor window.

Editor Insert/Component... opens up a dialog window with a list of all the components available for use in McStas. Selecting a component will display a description. Double-clicking will open up a dialog window allowing the entry of the values of all the parameters for the component (figure 4.5). See section 5.3 for details of the meaning of the different fields.

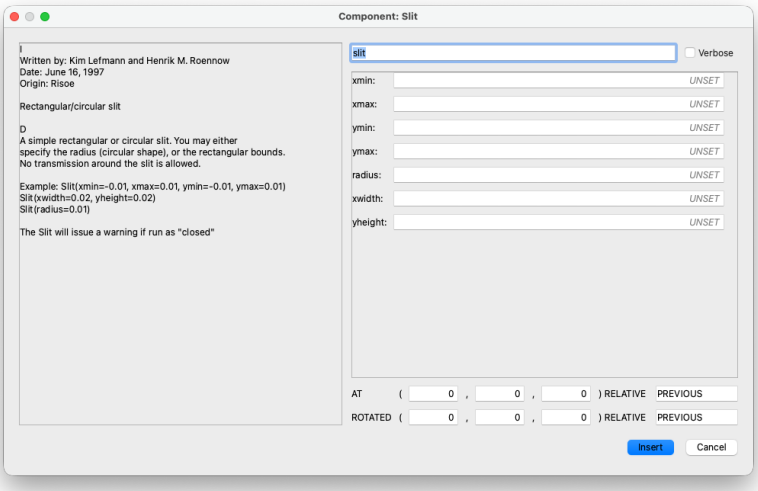


Figure 4.5.: Component parameter entry dialog.

The dialog will also pick up those of the users own components that are present in the current directory when `mcgui` is started. See section 5.7 for how to write components to integrate well with this facility.

Editor Insert/*Type* These menu entries give quick access to the entry dialog for the various component types available, i.e. Sources, Optics, Samples, Monitors, Misc, Contrib and Obsolete.

4.5.2. Running simulations on the commandline (mcrun)

The `mcrun` front-end provides a convenient command-line interface for running simulations with the same automatic compilation features available in the `mcgui` front-end. It also provides a facility for running a series of simulations while varying an input parameter.

The command

```
1 mcrun sim args ...
```

will compile the instrument definition *sim*`.instr` (if necessary) into an executable simulation *sim*`.out`. It will then run *sim*`.out`, passing the argument list *args*

The possible arguments are the same as those accepted by the simulations themselves as described in section 4.4, with the following extensions:

- The `-c` or `--force-compile` option may be used to force the recompilation of the instrument definition, regardless of file dates. This may be needed in case any

component definitions are changed (in which case `mcrun` does not automatically recompile), or if a new version of McStas has been installed.

- The `-p` *`file`* or `-param=`*`file`* option may be used to specify a file containing assignment of values to the input parameters of the instrument definition. The file should consist of specifications of the form *`name=value`* separated by spaces or line breaks. Multiple `-p` options may be given together with direct parameter specifications on the command line. If a parameter is assigned multiple times, later assignments override previous ones.

- The `-N` *`count`* or `-numpoints=`*`count`* option may be used to perform a series of *count* simulations while varying one or more parameters within specified intervals. Such a series of simulations is called a *scan*. To specify an interval for a parameter X, it should be assigned two values separated by a comma. For example, the command

```
1 mcrun sim.instr -N4 X=2,8 Y=1
```

  would run the simulation defined in `sim.instr` four times, with X having the values 2, 4, 6, and 8, respectively.

  After running the simulation, the results will be written to the file `mccode.dat` by default (see the `--optimise-file` option). This file contains one line for each simulation run giving the values of the scanned input variables along with the integrated intensity and estimated error in all monitors. This file can be plotted with any of the `mcplot` backends, see section 4.5.6.

- When performing a scan, the `-optimise-file` *`file`* option makes `mcrun` write the scan summary to *file* instead of the default `mccode.dat`.

- Two further scan modes are available besides the default one, in which all scanned parameters step together from their minimum to their maximum value. With `-M` or `-multi`, the cartesian product of all scanned parameters' points is run (a multi-dimensional "grid" scan); the point count of each parameter is then given as a comma-separated list to `-N`, in the order the parameters appear on the command line. For example

```
1 mcrun sim.instr -M -N3,5 X=2,8 Y=0,1
```

  runs $3 \times 5 = 15$ simulations. With `-L` or `-list`, each scanned parameter is instead given as an explicit comma-separated list of values (which also allows non-numerical values such as file names); lists of equal length are stepped through together, and combined with `-M` the cartesian product is run (the lists may then have different lengths). A parameter may also be given as *`min:delta:max`*, in which case the point count follows from the requested step size and the parameter may be freely mixed with explicit lists. The option `-seeds` scans over a range of random seeds (each non-zero). In a scan with `-seed`, point i uses the seed SEED$+1024\,i$ (without `-seed` the base seed is taken from the current time and logged).

- With `-autoplot`, `mcrun` opens a plotter on the generated data set when done (the back-end is chosen with `-autoplotter`, and `-invcanvas` requests an inverted canvas). With `-C` or `-c-lint`, the generated C code is only passed through a C linter (e.g. `cppcheck`, configured in `mccode_config.json`) and no simulation is run.

- When performing a scan, the `-d` *dir* and `-dir=`*dir* options make `mcrun` put all output in a newly created directory *dir*. Additionally, the directory will have subdirectories `1`, `2`, `3`,... containing all data files output from the different simulations. When the `-d` option is not used, no data files are written from the individual simulations (in order to save disk space).

The `-h` option will list valid options; see also Table 4.1 for the complete, current option list.

4.5.3. Interactive notebooks (mcstas-jupylab)

The `mcstas-jupylab` front-end offers an interactive, notebook-based alternative to editing an `.instr` file directly and running it via `mcgui`/`mcrun`. Given an instrument file:

```
1 mcstas-jupylab name.instr
```

it first runs `mcstas-pygen` (see above) to translate the instrument into an equivalent McStasScript Python model, writes that model into a notebook (*name*`_generated.ipynb`) built from a ready-made template (imports, and cells calling `show_diagram()`, `show_instrument()`, `backengine()` and the plotting helpers are already filled in), and opens it in JupyterLab[1]. From there the instrument can be inspected, modified and run cell by cell using McStasScript's own Python API, which is useful for exploratory work and for combining a simulation with other Python-based analysis in the same notebook.

Note that this generates a McStasScript model *once*, as a starting point; it is not kept in sync with later edits to the original `.instr` file, nor vice versa. See the McStasScript documentation [Mcsb] for the Python API itself.

4.5.4. GPU acceleration via OpenACC

McStas 3.x supports GPU-accelerated simulations through the OpenACC framework, enabling substantial speed-ups over CPU-only execution on compatible NVIDIA hardware.

Requirements

- **Compiler:** NVIDIA HPC SDK version 20.x or newer. The free Community Edition is sufficient.

[1] A working `jupyter lab` installation is required; it is not currently bundled with the standard conda-forge McStas package.

- **Hardware:** A CUDA-capable NVIDIA GPU with an up-to-date driver.
- **Platform:** GPU support is currently available on Linux only. Windows is not yet supported for OpenACC by NVIDIA; Windows users wishing to use GPU acceleration should do so via WSL 2 running a Linux distribution.

Running a GPU-accelerated simulation

Pass the `--openacc` flag to `mcrun`:

```
mcrun --openacc MyInstrument.instr -n 1e9
```

The OpenACC execution can be tuned with `--vecsize` (vector length), `--numgangs` (number of "gangs"), `--gpu_innerloop` (maximum number of particles per GPU kernel run; if smaller than the requested neutron count, the kernel is repeated) and `--funnel` (funnelled simulation flow, e.g. for mixed CPU/GPU execution), see Table 4.2 and Table 4.1.

Combined multi-core + GPU execution. It is also possible to use both CPU cores and the GPU simultaneously. Enable this by adding the following to the `OACCFLAGS` field of your `mccode_config.json`:

```
"OACCFLAGS": "-fast -Minfo=accel -acc=gpu,multicore
              -gpu=managed -DOPENACC -DMULTICORE"
```

Further information

GPU terminology specific to McStas/McXtrace 3 and detailed debugging tips are documented on the McCode wiki:

- GPU terminology table:
  `https://github.com/mccode-dev/McCode/wiki/McStas-McXtrace-3-and-GPU-terminology`
- GPU debugging tips:
  `https://github.com/mccode-dev/McCode/wiki/GPU-Debugging-tips`

4.5.5. Graphical display of simulations (mcdisplay)

The front-end `mcdisplay` is a graphical visualization tool, very useful for debugging. It presents a schematic drawing of the instrument definition, showing the position of the components and the paths of the simulated neutrons through the instrument. It is thus very useful for debugging a simulation, for example to spot components in the wrong position or to find out where neutrons are getting lost.

To use the `mcdisplay` front-end with a simulation, run it as follows:

```
1 mcdisplay sim args ...
```

where `sim` is the name of either the instrument source *sim*`.instr` or the simulation program *sim*`.out` generated by `mcstas`, and *args ...* are the normal command line arguments for the simulation, as explained above. The `-h` option will list valid options.

The drawing back-end may be selected by invoking one of the dedicated front-ends directly:

- `mcdisplay-pyqtgraph` (the default when running plain `mcdisplay`): interactive PyQtGraph 3D view.

- `mcdisplay-matplotlib`: static/interactive matplotlib 3D view. This is a thin wrapper script, so its options differ slightly from the default front-end: `--trace={1,2}` selects the classic (1) or new (2, default) drawing mode, `--backend=NAME` selects the matplotlib back-end (the hard-copy back-ends `pdf`, `pgf`, `ps` and `svg` save a file instead of opening a window), and the number of traced rays is capped at 100.

- `mcdisplay-webgl` and `mcdisplay-webgl-classic`: browser-based WebGL views. The former ships as a prebuilt web application served by a small local web server (only if its `dist/` folder is missing does the first run perform a one-time `npm install && npm run build`, which needs internet access). Both accept `-t`/`--trace N` (visualisation mode, default 2: boxes, cylinders, OFF geometry etc. are drawn as 3D shapes; 1 selects the legacy line approximations), `-d`/`--dirname DIR`, `--inspect COMP` (show only rays reaching component `COMP`) and `--nobrowse` (do not open a browser). `mcdisplay-webgl` also has `--timeout SEC` (the shutdown time of its local viewer web server, default 300 s), while `mcdisplay-webgl-classic` additionally accepts `--first COMP`, `--last COMP` (the first and last component of the zoomed range) and `--invcanvas`.

- `mcdisplay-plopp`: draws the instrument and the traced rays with the `plopp`/`scipp` Python packages (install e.g. with `conda install -c conda-forge plopp ipympl jupyterlab`). By default it opens an interactive 3D view in JupyterLab; with `-o FILE` it instead writes a static `png`, `pdf` or `svg` with 2D projections. Accepts `-n N` (number of rays, default 100), `-y` (forwarded to `mcrun`: use defaults for unset parameters) and the instrument parameters as `name=value`.

- `mcdisplay-cad`: exports the instrument geometry to a CAD file and requires the `cadquery` Python package. The format is selected with `-f`/`--format`: `step` (default), `stl`, `xml`, `vrml`, `gltf` or `vtkjs`. As no rays are needed for a geometry export, `--ncount` defaults to 0 here.

- `mcdisplay-mantid_xml`: exports an approximate Mantid Instrument Definition File (IDF) describing the detector geometry.

For instance, calling

```
1 mcdisplay-webgl BNL_H8.instr lambda=2.36
```

will open a WebGL view of the instrument in a browser tab. The `mcdisplay` front-end can also be run from the `mcgui` front-end.

The (default, PyQtGraph) front-end accepts the following options, amongst others – run `mcdisplay --help` for the complete, current list:

- `--default`: automatically use the instrument's default parameter values, without prompting.
- `--dirname=DIR`: override the output directory name used while tracing.
- `--inspect=`*comp*: only display neutron trajectories that reach the component named *comp*. This is useful when debugging, e.g. to hide neutrons absorbed upstream of the component of interest.
- `--invcanvas`: invert the canvas background from black to white (useful for printing or screenshots).
- `-n`/`--ncount=N`: number of neutron rays to trace (default 300).
- `--no-mpi`: simulate without MPI (forwards `--no-mpi` to `mcrun`).
- `--tof`: display in time-of-flight mode.

All variants run `mcrun --trace` underneath, so an instrument parameter may be given as *name*=*value* just as for `mcrun`. In the PyQtGraph viewer, `q` quits, `p` saves a PNG image, `s` saves an SVG image (not on Windows), `space` or `F5` shows the next ray, a click on a sub-plot zooms in on it (right-click to leave the zoomed view), and `h` or `F1` lists the components. A legacy Matlab/Octave variant, `mcdisplay-matlab`, also exists under `tools/matlab` (it falls back to the Python `mcdisplay` when neither Matlab nor Octave is found).

When debugging trajectories through a long or complex instrument, combining `--inspect` with a modest `--ncount` typically gives the clearest, fastest picture of what is happening around a particular component.

4.5.6. Plotting the results of a simulation (mcplot)

The front-end `mcplot` is a program that produces plots of all the monitors in a simulation, and it is thus useful to get a quick overview of the simulation results.

In the simplest case, the front-end is run simply by typing

```
1    mcplot
```

This will plot any simulation data stored in the current directory, which is where simulations store their results by default. If the `--dir` option has been used (see section 4.4), the name of the directory should be passed to mcplot, *e.g.* "`mcplot dir`". It is also possible to plot one single text (not binary) data file from a given monitor, passing its name to mcplot.

As with `mcdisplay` (section 4.5.5), the drawing back-end may be selected by invoking the dedicated front-end directly: `mcplot-pyqtgraph` (the default), `mcplot-matplotlib`, `mcplot-plopp` (plopp/scipp based, see below), or `mcplot-html` (renders an interactive, self-contained web page using D3.js, see section 4.4.2), or by setting the `MCSTAS_FORMAT` environment variable.

The back-ends share the simulation file or directory argument, and have a few options of their own. `mcplot-html` accepts `-n`/`--nobrowse` (do not open a browser), `-l`/`--log` (log scale), `--autosize` (expand the plots to the window size), `-o`/`--output FILE` (output `.html` file) and `-W`/`-H` (width and height of the plot frames). `mcplot-matplotlib` accepts `--log`, `--backend NAME`, `--format FMT` together with `--output FILE` (save as `pdf`, `png`, `eps`, `svg`, ... without opening a window) and `--html` (save as HTML via mpld3; Linux only), and `mcplot-pyqtgraph` accepts `--invcanvas` (white instead of black canvas). Both the matplotlib and PyQtGraph variants also take `-t`/`--test`, which only runs the data loader and prints the structure of the plot. A legacy Matlab/Octave/iFit variant, `mcplot-matlab`, exists under `tools/matlab`; it picks whichever of Matlab, iFit and Octave is available (falling back to the Python `mcplot`), and can export each monitor with `-png`, `-jpg`, `-fig`, `-eps` or `-pdf`.

`mcplot-plopp` opens the monitors of a simulation directory, `mccode.sim`, a single monitor file or a NeXus `mccode.h5` file in an interactive JupyterLab view built on `plopp`/`scipp` (install with e.g. `conda install -c conda-forge plopp ipympl jupyterlab`). With `-o FILE` it writes a static `png`, `pdf` or `svg` instead, `--log` selects a logarithmic intensity scale, and `--source`/`--sample` name the source and sample components when plotting Monitor_nD event data from NeXus files through McStasToX.

All plotters read the plain-text McStas/McCode data format (section 4.6); NeXus files are not currently read back by the plotters.

The `mcplot` front-end can also be run from the `mcgui` front-end.

The initial display shows plots for each detector in the simulation.

Comparing simulation results: mcplotdiff and mccoplot. Two companion tools share the loading and monitor-matching code of `mcplot` to compare simulation results (output directories, or single monitor files), matching monitors by output filename. Each exists in the same three rendering variants as `mcplot` (`-html`, `-matplotlib` and `-pyqtgraph`); the browser-based HTML variants are described here:

- `mcplotdiff-html a b` plots, for every monitor present with matching binning in both `a` and `b`, the difference `a - b` (with error-propagated error bars for 1D monitors, and a diverging blue/white/red colour scale for 2D monitors). This is useful to quickly spot the effect of a component or parameter change, an McStas version upgrade, or MPI vs. non-MPI results. By default, each difference is also written back out as a normal McStas/McCode-format data set (with an `mccode.sim` index), so the result directory can be reopened by any other McCode-format-aware plotter; pass `-D`/`--no-dat` to skip this. It always compares exactly two datasets.
- `mccoplot-html a b [c ...]` instead overlays the 1D monitors of *two or more* datasets on the same axes, for a direct visual comparison of curve shape and position

(only 1D monitors are supported; a monitor that is not a matching 1D monitor in every dataset is skipped with a warning). The compact legend letters `A`, `B`, `C`, ... are explained in an identity line under the title. Use `-L`/`--labels A,B,C` to set a short label per dataset, `-C`/`--colours c1,c2,...` to set the overlay colours, and `--no-legends` / `--no-titles` to leave out the on-plot legend respectively the in-plot title and identity line.

`mcplotdiff-html` accepts `-A LABEL_A`/`-B LABEL_B` to control the labels used for each input, and both tools accept `-o OUTPUT` to control the output directory name, `-n`/`--nobrowse`, `-l`/`--log` and `--autosize`; run either with `--help` for the complete option list. The matplotlib and PyQtGraph variants take the same dataset arguments and labelling options, plus the output options of the corresponding `mcplot` back-end. `mcplotdiff-html` and `mccoplot-html` are also used internally by `mcviewtest` (section 4.5) to compare test runs against a reference.

4.5.7. Plotting resolution functions (mcresplot)

The `mcresplot` front-end is used to plot the resolution function, particularly for triple-axis spectrometers, as calculated by the Res_sample component or TOF_Res_sample for time-of-flight instruments. It requires to have a Res_monitor (or TOF_Res_monitor) component further in the instrument description (at the detector position).

As of McStas 3.x, `mcresplot` is a pure Python tool built around a covariance-matrix method (rather than reading a resolution ellipsoid directly out of the simulation), following the approach implemented in the Takin2 package [Web20]. Neutron events collected by the Res_monitor are loaded with NumPy, the weighted covariance matrix of the $(\boldsymbol{Q}, \omega)$ four-vectors is computed and inverted to obtain the resolution matrix, and the resulting ellipsoids are drawn with `matplotlib` (2D projections and slices, plus a 3D view).

The `mcresplot` front-end is launched with the command

```
1 mcresplot outfile
```

Here, *outfile* is the name of a file output from a simulation using the Res_monitor component. Useful options include `--QEfile` (if the event file is in the $h\,k\,l\,E\,w$ format rather than the default $\mathbf{k}_i\,\mathbf{k}_f\,w_i\,w_f$ format), `--centreonQ` (center plots on the mean $\boldsymbol{Q}$ rather than the origin), `--save=file` (save the plots to a file instead of, or in addition to, showing them interactively), `--noplot` (compute and print the resolution matrices only, without opening a plot window) and `--tex` (use LaTeX for axis labels). Run `mcresplot --help` for the full, current option list.

The front-end will open a window displaying projections of the 4-dimensional resolution function $R(\boldsymbol{Q}, \omega)$, measured at a particular choice of $\boldsymbol{Q}$ and ω, see the component manual. The covariance matrix of the resolution function, the resolution along each projection axis and the resulting resolution matrix are also shown, as well as the instrument name and parameters used for the simulation.

To use the `mcresplot` front-end, a Python installation with NumPy and matplotlib is required (both are part of the standard McStas conda-forge environment).

4.5.8. Creating and viewing the library, component/instrument help and Manuals (mcdoc)

McStas provides an easy way to generate automatically an HTML help page about a given component or instrument, or the whole McStas library.

```
mcdoc
mcdoc searchterm
mcdoc file.comp
```

The first example generates an *index.html* catalog file using the available components and instruments (both locally, and in the McStas library), and browses it using the `BROWSER` environment variable (e.g. firefox, chromium, ...). Alternatively, if a search term or a *file.comp*/*file.instr* path is given, `mcdoc` will search within the library and open documentation for all matching entries. The search results are written to a separate page, `mcdoc_search.html`, next to the main library page (or in the temporary directory if that is read-only), so the main page is left untouched. Mathematics in component documentation is rendered locally with the MathJax copy installed with the documentation.

Additional options include `--install`/`-i` (regenerate the local installation-wide index), `--dir=DIR`/`-d` (add search results from a given directory), and `--verbose`/`-v` (print a parsing log). HTML is always written; `--md`/`-M` and `--tex`/`-T` also emit Markdown respectively LaTeX documentation files (`--all-formats`/`-A` for all three) — the LaTeX snippets of the component library that make up the Component Manual are produced this way. With `--in-repo` the output is placed next to the source files instead of in the installed documentation directory (and no master page is made), and `--outdir=DIR`/`-o` selects the output directory (in-repo use only). The options `--manual`/`-m`, `--comps`/`-c` and `--web`/`-w` will open the User Manual (this document), the Component Manual, and the McStas web site, respectively, all requiring `BROWSER` to be defined. Finally, the `--help` option will display the command help, as usual.

See section 5.7 for more details about the McDoc usage and header format. `mcdoc` is a pure Python tool, part of the standard McStas conda-forge package.

Online wiki. In addition to this manual, extensive and frequently updated documentation is maintained on the McCode GitHub wiki [Wik]:

`https://github.com/mccode-dev/McCode/wiki`

The wiki includes end-user guides, component development howtos, GPU acceleration information, and migration guides from McStas 2.x to 3.x.

4.5.9. Self-testing the installation (mctest, mcviewtest)

McStas ships with a Python self-test/benchmark harness, replacing the earlier `mcrun --test` command. `mctest` compiles and runs a selection (or all) of the example instruments found under the current installation(s), and stores each run's results (including any test-defined `%Test` reference values) in a timestamped output folder. Each test compiles

the instrument, displays it with a single particle, runs it, and compares the result with the target value recorded in the instrument header. Useful options include `--ncount` (default 10^6), `--seed` (default 1000; 0 randomizes), `--mpi`, `--no-mpi`, `--openacc` and `--nexus` (forwarded to `mcrun` for every tested instrument), `--lint` (only run the C linter, no simulations), `--config=LABEL` (test only a specific McCode installation/config), `--instr=REGEX` (test only matching instrument names), `--comp=COMP[,COMP...]` (test only instruments using any of the given components), `--local=DIR` (test instruments from `DIR` instead of the installation), `--limit=N` (test only the first N instruments per version), `--skipnontest` (skip instruments without an `%Test`/`%TestScan` line), `--strict` (fail instruments without `%Test`/`%TestScan` line(s) immediately) and `--permissive` (exit with status 0 even if tests fail), `--testdir=DIR` and `--suffix=SUFFIX` (where the results are written), `--compilemax`, `--runmax` and `--displaymax` (time limits in seconds) and `--noplots` (do not generate the overview plots); run `mctest --help` for the full list.

A test is defined in the instrument header by a `%Test:` line (older name `%Example:`), giving instrument parameters and the expected monitor value, *e.g.* `%Test: lambda=2.36 Detector: He3H_I=9.84026e-10`. A `%TestScan:` line (older name `%Scan:`) defines a parameter scan using the `mcrun` scan syntax, followed by a `{}`-enclosed list with one target value per scan point (it may span several header lines), *e.g.* `%TestScan: dBz=-0.0001,0.0001 -N41 -n1e5 Detector: detector_I={...}`. A `%Test:` always runs at the global `--ncount` (scaled by `--statfactor`); a scan uses its own `-n` if given, otherwise at most 10^5 per point, and runs its points in parallel (`--scan_split=auto`) unless MPI is used. By default a value (every point of a scan) passes when it is within 20 % of its target. With `--nsigma N` or `--pvalue P` it is instead accepted within the statistical tolerance of the combined error bars, using the error bar of the target when the test line gives it (*e.g.* `NAME_I=2.00752e+08 NAME_ERR=2.28872e+06`). `--statfactor F` scales the particle count of every test (*e.g.* 0.01 for quick runs, 100 for reference-quality ones) and `--noscans` skips the `%TestScan` tests. The statistics are described in more detail in `tools/Python/mctest-statistics.md`.

`mcviewtest` then generates an HTML report of one or more `mctest` runs found in a given folder (the current directory by default). When more than one run is present (e.g. a reference run and a run on a different platform, GPU vs. CPU, or a different McStas branch), it takes the oldest subfolder as the reference column (override with `--reflabel=LABEL`), and additionally runs `mcplotdiff-html` and `mccoplot-html` (section 4.5.6) between each other column's monitor output and the reference's, adding a `DIFF [vs ref]` link next to each such row in the report. These comparisons are cached on disk and generated in parallel (`--diffworkers`, each limited to `--diffmax` seconds), so re-running `mcviewtest` is fast (each test, and each scan, is one row of the report, coloured pass/fail by the rule `mctest` used for that result set; hovering a scan cell lists every point's value, target and percentage); use `--nodiff` to skip generating them entirely, or `--diff-errors-only` to only diff rows that already show a discrepancy (by default every valid row is diffed). `--nobrowse` keeps the finished report from being opened in a browser.

4.6. Data formats - Analyzing and visualizing the simulation results

To analyze simulation results, one uses the same tools as for analyzing experimental data, *i.e.* programs such as Python/NumPy, Matlab, IDL. The output files from simulations are usually simple text files containing headers and data blocks. If data blocks are empty they may be accessed referring to an external file indicated in the header.

Each data file contains information about the simulation, the instrument, the parameters used, and of course the signal, the estimated error on the signal, and the number of events used in each bin. Additionally, all data files indicate their first moment (mean value) and second moment (half width) in the 'statistics' field.

The available data formats are the legacy McStas (McCode) format, and the HDF/NeXus binary when the needed libraries are installed.

In order for the user to choose the data format, we recommend to set it using the `--format=FORMAT` or alternatively *via* the MCSTAS_FORMAT environment variable, which will also make the front-end programs able to import and plot data and instrument consistently (see Section 4.4).

Note that the neutron event counts in detectors are typically not very meaningful except as a way to measure the performance of the simulation. Use the simulated intensity instead whenever analyzing simulation data.

4.6.1. The McStas/McCode text format

The McStas original format, sometimes still referred to by its historical name "PGPLOT format" (the plotting package used in McStas 1.x/2.x), is simply columns of ASCII text that most programs should be able to read.

One-dimensional histogram monitors (time-of-flight, energy) write one line for each histogram bin. Each line contains a number identifying the bin (*i.e.* the time-of-flight) followed by three numbers: the simulated intensity, an estimate of the statistical error as explained in section 3.2.1, and the number of neutron events for this bin.

Two-dimensional histogram monitors (position sensitive detectors) output M lines of N numbers representing neutron intensities, where M and N are the number of bins in the two dimensions. The two-dimensional monitors also store the error estimates and event counts as additional matrices.

Single-point monitors output the neutron intensity, the estimated error, and the neutron event count as numbers on the terminal. (The results from a series of simulations may be combined in a data file using the `mcrun` front-end as explained in section 4.5.2).

When using one- and two-dimensional monitors, the integrated intensities are written to terminal as for the single-point monitor type, supplementing file output of the full one- or two-dimensional intensity distribution. Both one- and two-dimensional monitor output by default start with a header of comment lines, all beginning with the '`#`' character. This header gives such information as the name of the instrument used in the simulation, the values of any instrument parameters, the name of the monitor component for this data file, *etc.*

In addition to the files written for each one- and two-dimensional monitor component, another file (by default named `mccode.sim`) is also created. This file is in a special McStas ASCII format. It contains all available information about the instrument definition used for the simulation, the parameters and options used to run the simulation, and the monitor components present in the instrument. It is read by the `mcplot` front-end (see section 4.5.6). This file stores the results from single monitors, but by default contains only pointers (in the form of file names) to data for one- and two-dimensional monitors. By storing data in separate files, reading the data with programs that do not know the special McStas file format is simplified.

4.6.2. NeXus format

The NeXus format [Nex] is a platform independent HDF binary data file. To have McStas use it

1. the HDF5 library must have been installed (libhdf5 and headers)

2. the compilation of instruments must be done with the `-DUSE_NEXUS -lhdf5` flag (see Section 5.3.8). This is automated with the `mcrun` tool (Section 4.5.2).

All results are saved in a single file, containing 'groups' of data. To view such files, install and use HDFView (or alternatively HDFExplorer). This Java viewer can show content of all detectors, including metadata (attributes). Basic detector images may also be generated.

4.7. Using MPI for parallel computing

Parallelizing a computation is in general possible when dependencies between each computation are not too strong. The situation of McStas is ideal since each neutron ray can be simulated without interfering with other simulated neutron rays. Therefore each neutron ray can be simulated independently on a set of computers.

When computing N neutron rays with p computers, each computer will simulate $\frac{N}{p}$ neutrons. As a result there will be $p \cdot \frac{N}{p} = N$ neutrons simulated. As a result, McStas generates two kinds of data sets:

- intensity measurements, internally represented by three values (p_0, p_1, p_2) where p_0, p_1, p_2 are additive. Therefore the final value of p_0 is the sum of all local value of p_0 computed on each node. The same rule applies for p_1 and p_2. The evaluation of the intensity errors σ is performed using the final p_0, p_1, and p_2 arrays (see Section 3.2.1).

- event lists: the merge of events is done by concatenation

McStas provides two methods in order to distribute computations on many computers.

- when using an homogeneous computer cluster, each simulation (including scan steps) may be computed in parallel using MPI. We recommend this method on clusters (see section 4.7.1).

- `mcrun`'s built-in scan-splitting (`--scan_split=N`, see Table 4.1) distributes the individual steps of a parameter scan across `N` local CPU cores/threads (`auto`, or 0, uses the number of cores minus one), without requiring MPI. It is ignored, with a warning, when an explicit multi-process `--mpi` is given. This is a lightweight alternative on a single multi-core machine.

All of these methods can be used, when available, from `mcgui`.

4.7.1. Parallel computing (MPI)

The MPI support requires that an MPI implementation (MPICH or OpenMPI are both known to work) is installed on a set of nodes, together with a properly configured passwordless `ssh` between nodes if running across more than one machine.

There are 2 methods for using MPI

- Basic usage requires to compile and run the simulation by hand (mpicc, mpirun).

- A much simpler way is to use `mcrun -c --mpi=NB_CPU ...` which will recompile and run with MPI support.

- The McGUI interface also supports MPI from within the Run Dialog.

The MPI support is especially suited on clusters.

Requirements and limitation (MPI)

To use MPI you will need

1. A working MPI installation (MPICH or OpenMPI) on all nodes, and a McStas installation accessible from all nodes (e.g. on a shared filesystem, or installed identically on each node).

2. If running across more than one machine, `ssh` access between nodes without a password (e.g. using `ssh-keygen` and `ssh-copy-id`), and a machine list file (see `--machines` in Table 4.1).

3. Signals are *not* supported while simulating with MPI (since asynchronous events cannot be easily transmitted to all nodes). This means it is not possible to cancel an on-going computation. However, simulation scans can be interrupted as soon as the on-going computation step ends.

MPI Basic usage

To enable parallel computation, compile `mcstas`-generated C code with `mpicc` with the flag `-DUSE_MPI` and run it using the wrapper of your MPI implementation (`mpirun` for mpich or lammpi) :

```
# generate a C-source file [sim.c]
mcstas sim.instr

# generate an executable with MPI support [sim.mpi]
mpicc -DUSE_MPI -o sim.mpi sim.c

# execute with parallel processing over <N> computers
# here you have to list the computers you want to use
# in a file [machines.list] (using mpich implementation)
# (refer to MPI documentation for a complete description)
mpirun -machinefile machines.list -n <N> \
       ./sim.mpi <instrument parameters>
...
```

If you don't want to spread the simulation, run it as usual:

```
./sim.mpi <instrument parameters>
```

4.7.2. McRun options for MPI

The two relevant `mcrun` options are (see Table 4.1):

- `--mpi=<number>`: tells `mcrun` to use MPI, and to spread the simulation over `<number>` nodes
- `--machines=<file>`: defines a text file where the nodes which are to be used for parallel computation are listed, one node per line.

When available, the MPI option will show up in the `mcgui` Run dialog. Specify the number of nodes required.

Suppose you have four machines named `node1` to `node4`. A typical machine list file, `machines.list` looks like :

```
node1
node2
node3
node4
```

You can then spread a simulation `sim.instr` using `mcrun` :

```
mcrun -c --mpi=4 --machines=machines.list \
      sim.instr <instrument parameters>
```

Warning: when using `mcrun` with MPI, be sure to recompile your simulation with MPI support (see `-c` flag of `mcrun`): a simulation compiled without MPI support cannot be used with MPI, whereas a simulation compiled with MPI support can be used without MPI.

4.7.3. McStas/MPI Performance

Theoretically, a computation which lasts T seconds on a single computer, should lasts at least $\frac{T}{p}$ seconds when it is distributed over p computers. In practice, there will be overhead time due to the split and merge operations.

- the split is immediate: constant time cost $\mathcal{O}(1)$
- the merge is at worst linear against the number of computers:
  - linear time cost : $\mathcal{O}(p)$ when saving an event list
  - logarithmic time cost: $\mathcal{O}(\log p)$ when not saving an event list

The efficiency of McStas using MPI has been tested on large clusters, up to 500 nodes. The computation time decreases in the same proportion as the number of nodes, showing an ideal efficiency. However, a small overhead may appear depending on the cluster internal network load, which may be estimated at most of about 10-20 s. This overhead comes from the spread and the fusion of the computations. For instance, spreading a computation implies often an `rsh` or and `ssh` session to be opened on every node. To reach the best efficiency, the computation time should not be lower than 30 seconds, or the overhead time may become significant compared to total time.

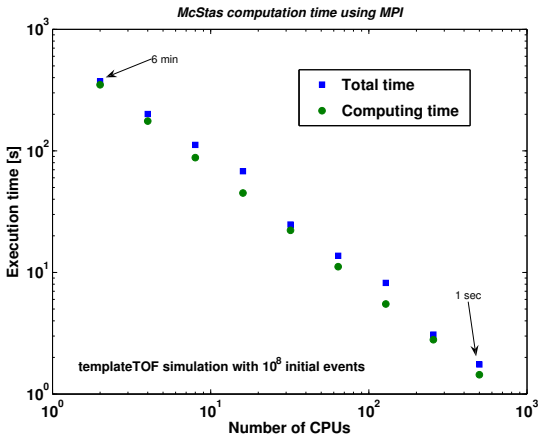


Figure 4.6.: McStas/MPI execution time as a function of computing nodes, with *templateTOF* instrument and 1e8 initial neutron events. Tests performed on Lonestar@TACC (US Teragrid, 2008).

4.7.4. MPI Bugs and limitations

- Some header of output files might contain minor errors.
- The computation split does not take into account the speed or the load of nodes: the overall time of a distributed computation is forced by the slowest node; for optimal performance, the "cluster" should be homogeneous.
- Interacting with a running simulation (USR1 and USR2 signals) is disabled with MPI.

5. The McStas kernel and meta-language

Instrument definitions are written in a special McStas meta-language which is translated automatically by the compiler `mcstas` into a C program which is in turn compiled to an executable that performs the simulation. The meta-language is custom-designed for neutron scattering and serves two main purposes: (i) to specify the interaction of a single neutron ray with a single optical component, and (ii) to build a simulation by constructing a complete instrument from individual components.

For maximum flexibility and efficiency, the meta-language is based on C. Instrument geometry, propagation of neutrons between the different components, parameters, data input/output etc. is handled in the meta-language and by the compiler `mcstas`. Complex calculations are written in C embedded in the meta-language description of the components. However, it is possible to set up an instrument from existing components and run a simulation without writing a single line of C code, working entirely in the meta-language.

Apart from the meta-language, McStas also includes a number of C library functions and definitions that are useful for neutron ray-tracing simulations. The definitions available for component developers are listed in appendix B. The list includes functions for

- Computing the intersection between a flight-path and various objects (such as planes, cylinders, boxes and spheres)
- Functions for generating random numbers with various distributions
- Functions for reading or writing information from/to data files
- Convenient conversion factors between relevant units, etc.

The McStas meta-language was designed to be readable, with a verbose syntax and explicit mentioning of otherwise implicit information. The recommended way to get started with the meta-language is to start by looking at the examples supplied with McStas, modifying them as necessary for the application at hand.

5.1. Notational conventions

Simulations generated by McStas use a semi-classical description of the neutron rays to compute the neutron trajectory through the instrument and its interaction with the different components. The effect of gravity is taken into account either in particular components (e.g. `Guide_gravity`), or more generally when setting an execution flag

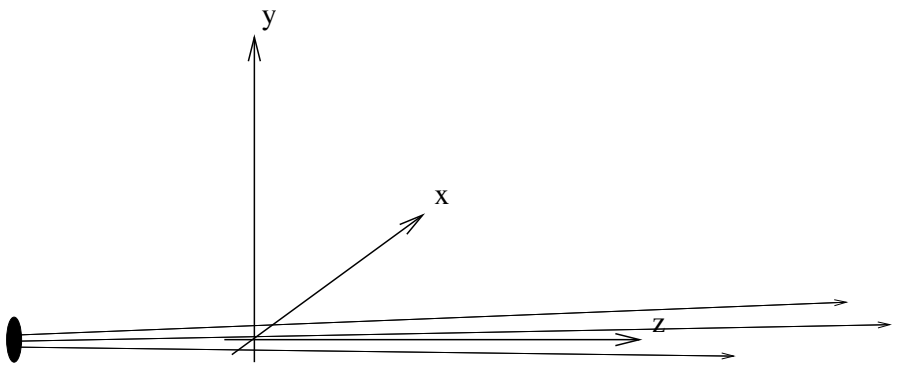


Figure 5.1.: conventions for the orientations of the axes in simulations.

(`-g`) to perform gravitation computation. This latter setting is only an approximation and may produce wrong results with some components.

An instrument consists of a list of components through which the neutron ray passes one after the other. The order of components is thus significant since `mcstas` does not automatically check which component is the next to interact with the neutron ray at a given point in the simulation. Note that in case of a negative propagation time from one component to the next, the neutron ray is by default *absorbed* as this is often an indication of unphysical conditions.

The instrument is given a global, absolute coordinate system. In addition, every component in the instrument has its own local coordinate system that can be given any desired position and orientation (though the position and orientation must remain fixed for the duration of a single simulation). By convention, the z axis points in the direction of the beam, the x axis is perpendicular to the beam in the horizontal plane pointing left as seen from the source, and the y axis points upwards (see figure 5.1). Nothing in the McStas metalanguage enforces this convention, but if every component used different conventions the user would be faced with a severe headache! It is therefore necessary that this convention is followed by users implementing new components.

In the instrument definitions, units of length (*e.g.* component positions) are given in meters and units of angles (*e.g.* rotations) are given in degrees.

The state of the neutron is given by its position (x, y, z) in meters, its velocity (v_x, v_y, v_z) in meters per second, the time t in seconds, and the three spin parameters (s_x, s_y, s_z), and finally the neutron weight p described in 3.

5.2. Syntactical conventions

Comments follow the C and C++ syntax

```
1 /* C style comment */
2 // C++ style comment
```

Keywords are not case-sensitive, for example "`DEFINE`", "`define`", and "`dEfInE`" are all equivalent. However, by convention we always write keywords in uppercase to distinguish them from identifiers and C language keywords. In contrast, McStas identifiers (names), like C identifiers and keywords, *are* case sensitive, another good reason to use a consistent case convention for keywords. All McStas keywords are reserved, and thus should not be used as C variable names. The list of these reserved keywords is shown in table 5.1.

It is possible, and usual, to split the input instrument definition across several different files. For example, if a component is not explicitly defined in the instrument, `mcstas` will search for a file containing the component definition in the standard component library (as well as in the current directory and any user-specified search directories, see section 4.3.2). It is also possible to explicitly include another file using a line of the form

```
1     %include "file"
```

Beware of possible confusion with the C language "`#include`" statement, especially when it is used in C code embedded within the McStas meta-language. Files referenced with "`%include`" are read when the instrument is translated into C by `mcstas`, and must contain valid McStas meta-language input (and possibly C code). Files referenced with "`#include`" are read when the C compiler generates an executable from the generated C code, and must contain valid C.

Embedded C code is used in several instances in the McStas meta-language. Such code is copied by `mcstas` into the generated simulation C program. Embedded C code is written by putting it between the special symbols % and %, as follows:

```
1 %{
2 // Embedded C code...
3 %}
```

The %{ and %} must appear on a line by themselves (do not add comments after). Additionally, if a "`%include`" statement is found *within* an embedded C code block, the specified file will be included from the 'share' directory of the standard component library (or from the current directory and any user-specified search directories) as a C library, just like the usual "`#include`" *but only once*. For instance, if many components require to read data from a file, they may all ask for "`%include "read_table-lib"`" without duplicating the code of this library. If the file has no extension, both `.h` and `.c` files will be searched and included, otherwise, only the specified file will be imported. The McStas'run-time' shared library is included by default (equivalent to "`%include "mcstas-r"`" in the `DECLARE` section). For an example of `%include`, see the monitors/Monitor_nD component. See also section 5.4 for insertion of full instruments in instruments (instrument concatenation).

If the instrument description compilation fails, check that the keywords syntax is correct, that no semi-colon (;) sign is missing (e.g. in C blocks and after an ABSORB macro), and there are no name conflicts between instrument and component instances variables.

Keyword	Scope	Meaning
ABSOLUTE	I	Indicates that the AT and ROTATED keywords are in the absolute coordinate system.
AT	I	Indicates the position of a component in an instrument definition.
COPY	I	Duplicate a previous component *instance* in the TRACE section (see section 5.4.3).
CPU	I	Forces a specific component instance to run on CPU only, even in an otherwise GPU/OpenACC-enabled instrument (implies DEPENDENCY "-DFUNNEL").
DECLARE	I,C	Declares C internal variables.
DEFINE	I,C	Starts an INSTRUMENT or COMPONENT definition.
DEFINITION	C	Defines component parameters that are constants (#define).
DEPENDENCY	C,I	Indicates any library dependency required to create the instrument.
DISPLAY	C	Alias for MCDISPLAY.
END	I,C	Ends the instrument or component definition.
SPLIT	I	Enhance incoming statistics by event repetition.
EXTEND	I,C	Extends a component TRACE section (plug-in), or extends an inherited section (see INHERIT).
FINALLY	I,C	Embeds C code to execute when simulation ends.
GROUP	I	Defines an exclusive group of components.
%include	I,C	Imports an instrument part, a component or a piece of C code (when within embedded C).
INHERIT	C	Derives a component (definition, or a single section) from an existing component (see section 5.6).
INITIALIZE	I,C	Embeds C code to be executed when starting. Alias: INITIALISE.
ITERATE	I	Defines iteration counter for JUMP.
JUMP	I	Iterative (loops) and conditional jumps.
MCDISPLAY	C	Embeds C code to display component geometry. Alias: DISPLAY.
METADATA	I,C	Attaches an arbitrary, typed, named block of text (e.g. an IDF/CIF/JSON fragment) to a component definition or instance, retrievable via mcrun -meta-* (see section 5.4.7).
MYSELF	I	Refers to the current component instance (COPY, JUMP).
NOACC	C	Marks a component class as CPU-only (never compiled for GPU/OpenACC).
NEXT	I	Refers to a following component instance (JUMP).
OUTPUT	C	Defines internal variables to be public and protected symbols (usually all global variables and functions of DECLARE). Alias: PRIVATE.
PARAMETERS	C	Defines a class of component parameter (DEFINITION, SETTING).
PREVIOUS	I,C	Refers to a previous component position/orientation/instance.
RELATIVE	I	Indicates that the AT and ROTATED keywords are relative to an other component.
REMOVABLE	I	Indicates that this component will be removed when the instrument is inserted into an other one using the %include keyword.
ROTATED	I	Indicates the orientation of a component in an instrument definition.
SAVE	I,C	Embedded C code to execute when saving data.
SEARCH	I,C	Appends an additional instrument/component search directory (optionally computed by an external SHELL command), see section 5.3.3.

5.3. Writing instrument definitions

The purpose of the instrument definition is to specify a sequence of components, along with their position and parameters, which together make up an instrument. Each component is given its own local coordinate system, the position and orientation of which may be specified by its translation and rotation relative to another component. An example is given in section 5.3.13 and some additional examples of instrument definitions can be found on the McStas web-page [Mcsa] and in the `example` directory.

As a summary, the usual grammar for instrument descriptions is

```
1 DEFINE INSTRUMENT name(parameters)
2 DECLARE C_code
3 INITIALIZE C_code
4 TRACE components
5 {FINALLY C_code}
6 END
```

5.3.1. The instrument definition head

```
1 DEFINE INSTRUMENT name (a_1, a_2, ...)
```

This marks the beginning of the definition. It also gives the name of the instrument and the list of instrument parameters. Instrument parameters describe the configuration of the instrument, and usually correspond to setting parameters of the components, see section 5.5. A motor position is a typical example of an instrument parameter. The input parameters of the instrument constitute the input that the user (or possibly a front-end program) must supply when the generated simulation is started.

By default, the parameters will be floating point numbers, and will have the C type `double` (double precision floating point). The type of each parameter may optionally be declared to be `int` for the C integer type or `char *` for the C string type. The name `string` may be used as a synonym for `char *`, and floating point parameters may be explicitly declared using the name `double`. The following example illustrates all possibilities:

```
1 DEFINE INSTRUMENT test(d1, double d2, int i, char *s1, string s2)
```

Here `d1` and `d2` will be floating point parameters of C type `double`, `i` will be an integer parameter of C type `int`, and `s1` and `s2` will be string parameters of C type `char *`. The parameters of an instrument may be given default values. Parameters with default values are called *optional parameters*, and need not be given an explicit value when the instrument simulation is executed. When executed without any parameter value in the command line (see section 4.4), the instrument asks for all parameter values, but pressing the `Return` key selects the default value (if any). When used with at least one parameter value in the command line, all non specified parameters will have their value set to the default one (if any). A parameter is given a default value using the syntax "*param*`=`*value*". For example

```
DEFINE INSTRUMENT test(d1= 1, string s2="hello")
```

Here `d1` and `d2` are optional parameters and if no value are given explicitly, "1" and "hello" will be used.

Optional parameters can greatly increase the convenience for users of instruments for which some parameters are seldom changed or of unclear signification to the user. Also, if all instrument parameters have default values, then the simple command `mcdisplay test.instr` will show the instrument view without requesting any other input, which is usually a good starting point to study the instrument design.

A numeric instrument parameter may optionally carry a documentation-only unit annotation, using a slash followed by a quoted string:

```
DEFINE INSTRUMENT test(double lambda/"AA" = 4.0, double L2/"m")
```

The unit string is purely informational (it is not converted or checked at run time); it is picked up by `mcdoc` and `mcgui` to annotate the parameter in generated documentation and entry dialogs, so it is good practice to use it for any parameter whose unit is not obvious from its name.

5.3.2. The `DEPENDENCY` line

```
DEPENDENCY "-lLIB1 -lLIB2 .."
```

If you make use of external library calls in your instrument, you may indicate the dependency list to be used for the compilation of the instrument, as a single DEPENDENCY keyword and a string argument, all on a single line such as:

```
DEPENDENCY "-lgsl -lgslcblas"
```

The concatenation of all found dependencies from the instrument, but also from all used components, will be printed to the terminal (stdout) as

```
CFLAGS=-lgsl -lgslcblas
```

This line is interpreted by `mcrun` (see section 4.5.2) and passed to the compiler. It must be indicated just before the `DECLARE` section. This line is optional, and if omitted you may need to manually link the instrument with the libraries and add the proper library names to the MCSTAS_CFLAGS or CFLAGS environment variables.

5.3.3. The `SEARCH` line

```
SEARCH "path/to/extra/components"
```

Appends an additional directory to the component/instrument search path (complementing the `-I` command line switch, see section 4.3.2). This is useful for instrument files that depend on a component library kept outside the standard McStas installation, without

requiring users to remember to pass `-I` manually. Like `DEPENDENCY`, `SEARCH` may appear at instrument level (before `DECLARE`) or inside a component definition.

A second form computes the path (or a newline-separated list of paths) by running an external command:

```
SEARCH SHELL "pkg-config --variable=compdir my-neutron-lib"
```

Each line printed by the command on its standard output is added as a search directory. See also `SHELL` below.

5.3.4. The `SHELL` line

```
SHELL "command to execute"
```

Executes an arbitrary shell command *before* code generation (i.e. as `mcstas` parses the file), printing its output to the terminal. This may be used for instance to auto-generate a data file or a piece of included C code as part of the build process. The compiler aborts if the command returns a non-zero exit status. Use with caution, since this executes arbitrary code on the machine running `mcstas`– do not `%include` untrusted instrument files containing `SHELL` lines.

5.3.5. The `DECLARE` section

```
DECLARE
%{
  // C declarations of global variables etc. ...
%}
```

This gives C declarations that may be referred to in the rest of the instrument definition. A typical use is to declare global variables or small functions that are used elsewhere in the instrument. The `%include ''file''` keyword may be used to import a specific component definition or a part of an instrument. Variables defined here are global, and may conflict with internal McStas variables, specially symbols like `x,y,z,sx,sy,sz,vx,vy,vz,t` and generally all names starting with `mc` should be avoided. If you can not compile the instrument, this may be the reason. The `DECLARE` section is optional.

5.3.6. The `USERVARS` section

```
USERVARS
%{
  double my_per_neutron_flag;
%}
```

`USERVARS` declares additional fields that are appended to *every* neutron ray's state (alongside the built-in `x,y,z,vx,vy,vz,t,sx,sy,sz,p`), rather than being ordinary global `DECLARE` variables. It may appear at instrument level (as here) or inside a component

definition, and both Instrument- and Component-level `USERVARS` end up as fields of the same per-neutron state struct.

This matters for two reasons:

- **Correctness with SPLIT/MPI/GPU:** a global `DECLARE` variable is shared mutable state across all in-flight neutron rays. On GPU/OpenACC (section 4.5.4 in the User Manual) or when many neutron rays are processed in parallel batches, this is unsafe – writes from one ray's computation can be read by another. A `USERVARS` field, by contrast, travels with its own neutron ray and is therefore safe to read/write from `EXTEND` blocks and component `TRACE` sections regardless of execution model.
- **Convenience:** it replaces the older pattern of communicating a per-ray flag between components via a global variable set in one `EXTEND` block and read in a later `WHEN` condition (see section 5.4.4), which is exactly the kind of global mutable state that is fragile under SPLIT and unsafe under GPU/parallel execution.

USERVARS fields are referred to directly by name from any `TRACE`, `EXTEND`, `WHEN`, or `MCDISPLAY` code, exactly like the built-in neutron state fields. The section is optional, and may be used at both instrument and component level in the same simulation.

5.3.7. The `INITIALIZE` section

```
1 INITIALIZE
2 %{
3 // C initializations.
4 %}
```

This gives code that is executed when the simulation starts. This section is optional. Instrument setting parameters may be modified in this section (e.g. doing tests or automatic settings).

5.3.8. The `NEXUS` extension

The NeXus format [Nex] requires to link the simulation to the HDF5 library, which must have been pre-installed. Preferably, McStas should have been installed with the `./configure --with-nexus` on Unix/Linux systems. To activate the NeXus output, the compilation of the instrument must be done with flag `-DUSE_NEXUS -lhdf5`. The resulting executable is no longer portable.

The default NeXus format is HDF5 with compression.

You may choose the name of the output file with the `-f filename` option from the instrument executable or `mcrun` (see Sections 4.4, 4.5.2 and Table 4.2).

Then, the output format is chosen as usual with the `--format=NeXus` option when launching the simulation. All output files are stored in the output *filename*, as well as the instrument description itself. Other formats are still available. When run on a distributed system (e.g. MPI), detectors are gathered, but list of events (see e.g. component Virtual_output) are stored as one data set per node.

5.3.9. The TRACE section

As a summary, the usual grammar for component instances within the instrument TRACE section is

```
COMPONENT name = comp(parameters)
  AT (...) [RELATIVE [reference|PREVIOUS] | ABSOLUTE]
 {ROTATED {RELATIVE [reference|PREVIOUS] | ABSOLUTE} }
```

The TRACE keyword starts a section giving the list of components that constitute the instrument. Components are declared like this:

```
COMPONENT name = comp(p_1 = e_1, p_2 = e_2, ...)
```

This declares a component named *name* that is an instance of the component definition named *comp*. The parameter list gives the setting and definition parameters for the component. The expressions $e_1, e_2, \ldots$ define the values of the parameters. For setting parameters arbitrary ISO-C expressions may be used, while for definition parameters only *constant* numbers, strings, names of instrument parameters, or names of C identifiers are allowed (see section 5.5.1 for details of the difference between definition and setting parameters). To assign the value of a general expression to a definition parameter, it is necessary to declare a variable in the DECLARE section, assign the value to the variable in the INITIALIZE section, and use the variable as the value for the parameter.

The McStas program takes care to rename parameters appropriately in the output so that no conflicts occur between different component definitions or between component and instrument definitions. It is thus possible (and usual) to use a component definition multiple times in an instrument description.

Beware about variable type conversion when setting numerical parameter values, as in `p1=12/1000`. In this example, the parameter `p1` will be set to 0 as the division of the two integers is indeed 0. To avoid that, use explicitly floating type numbers as in `p1=12.0/1000`.

The compiler `mcstas` will automatically search for a file containing a definition of the component if it has not been declared previously. The definition is searched for in a file called "*name*`.comp`". See section 4.3.2 for details on which directories are searched. This facility is often used to refer to existing component definitions in standard component libraries. It is also possible to write component definitions in the main file before the instrument definitions, or to explicitly read definitions from other files using `%include` (not within embedded C blocks).

The physical position of a component is specified using an AT modifier following the component declaration:

```
  AT (x,y,z) RELATIVE name
```

This places the component at position (x, y, z) in the coordinate system of the previously declared component *name*. Placement may also be absolute (not relative to any component) by writing

```
  AT (x,y,z) RELATIVE ABSOLUTE
```

Any C expression may be used for x, y, and z. The `AT` modifier is required. Rotation is achieved similarly by writing

```
ROTATED (phi_x,phi_y,phi_z) RELATIVE name
```

This will result in a coordinate system that is rotated first the angle ϕ_x (in degrees) around the x axis, then ϕ_y around the y axis, and finally ϕ_z around the z axis. Rotation may also be specified using `ABSOLUTE` rather than `RELATIVE`. If no rotation is specified, the default is $(0, 0, 0)$ using the same relative or absolute specification used in the `AT` modifier. We *strongly* recommend to apply all rotations of an instrument description on Arm class components only, acting as goniometers, and position the optics on top of these. This usually makes it much easier to orient pieces of the instrument, and avoid positioning errors.

The *position* of a component is actually the origin of its local coordinate system. Usually, this is used as the input window position (e.g. for guide-like components), or the center position for cylindrical/spherical components.

The `PREVIOUS` keyword is a generic name to refer to the previous component in the simulation. Moreover, the `PREVIOUS(n)` keyword will refer to the n-th previous component, starting from the current component, so that `PREVIOUS` is equivalent to `PREVIOUS(1)`. This keyword should be used after the `RELATIVE` keyword, but not for the first component instance of the instrument description.

```
AT (x,y,z) RELATIVE PREVIOUS
ROTATED (phi_x, phi_y, phi_z) RELATIVE PREVIOUS(2)
```

Invalid `PREVIOUS` references will be assumed to be absolute placement.

The order and position of components in the `TRACE` section does not allow components to overlap, except for particular cases (see the `GROUP` keyword below). Indeed, many components of the McStas library start by propagating the neutron event to the beginning of the component itself. Anyway, when the corresponding propagation time is found to be negative (*i.e.* the neutron ray is already *after* or *aside* the component, and has thus passed the 'active' position), the neutron event is ABSORBed, resulting in a zero intensity and event counts after a given position. The number of such removed neutrons is indicated at the end of the simulation. Getting such warning messages is an indication that either some components overlap, or some neutrons are getting outside of the simulation, for instance this usually happens after a monochromator, as the non-reflected beam is indeed lost. A special warning appears when no neutron ray has reached some part of the simulation. This is usually the sign of either overlapping components or a very low intensity.

For experienced users, we recommend as well the usage of the `WHEN` and `EXTEND` keywords, as well as other syntax extensions presented in section 5.4 below.

5.3.10. The `SAVE` section

```
SAVE
%{
```

```
3   // C code to execute each time a temporary save is required...
4 %}
```

This gives code that will be executed when the simulation is requested to save data, for instance when receiving a USR2 signal (on Unix systems), or using the `Progress_bar` component with intermediate savings. It is also executed when the simulation ends. This section is optional.

5.3.11. The `FINALLY` section

```
1 FINALLY
2 %{
3    // C code to execute at end of simulation
4 %}
```

This gives code that will be executed when the simulation has ended. When existing, the `SAVE` section is first executed. The `FINALLY` section is optional. A simulation may be requested to end before all neutrons have been traced when receiving a TERM or INT signal (on Unix systems), or with Control-C, causing code in `FINALLY` to be evaluated.

5.3.12. The end of the instrument definition

The end of the instrument definition must be explicitly marked using the keyword

```
1 END
```

5.3.13. Code for the instrument `vanadium_example.instr`

A commented instrument definition taken from the `examples` directory is here shown as an example of the use of McStas.

5.4. Writing instrument definitions - complex arrangements and syntax

In this section, we describe some additional ways to build instruments using groups, code extension, conditions, loops and duplication of components.

As a summary, the nearly complete grammar definition for component instances within the instrument TRACE section is:

```
1 {REMOVABLE} {CPU} {SPLIT} COMPONENT name = comp(parameters) {WHEN condition
      }
2   AT (...) [RELATIVE [reference|PREVIOUS] | ABSOLUTE]
3   {ROTATED {RELATIVE [reference|PREVIOUS] | ABSOLUTE} }
4   {GROUP group_name}
5   {EXTEND C_code}
6   {JUMP [reference|PREVIOUS|MYSELF|NEXT] [ITERATE number_of_times | WHEN
        condition] }
```

```
7   {METADATA type name C_code}
```

The `REMOVABLE` keyword is documented in section 5.4.1 below, `CPU` in section 5.5.2, and `METADATA` in section 5.4.7.

5.4.1. Embedding instruments in instruments TRACE

The `%include` insertion mechanism may be used within the TRACE section, in order to concatenate instruments together. This way, each DECLARE, INITIALIZE, SAVE, and FINALLY C blocks, as well as instrument parameters from each part are concatenated. The TRACE section is made of inserted COMPONENTS from each part. In principle, it is then possible to write an instrument as:

```
1 DEFINE concatenated()
2 TRACE
3
4 %include "part1.instr"
5 %include "part2.instr"
6
7 END
```

where each inserted instrument is a valid full instrument. In order to avoid some components to be duplicated - e.g. Sources from each part - a special syntax in the TRACE section

```
1 REMOVABLE COMPONENT a=...
```

marks the component *a* as removable when inserted. In principle, inserted instruments may themselves use `%include`.

5.4.2. Groups and component extensions - GROUP - EXTEND

It is sometimes desirable to slightly modify an existing component of the McStas library. One would usually make a copy of the component, and extend the code of its `TRACE` section. McStas provides an easy way to change the behavior of existing components in an instrument definition without duplicating files, using the `EXTEND` modifier

```
1 EXTEND
2 %{
3 // C code executed after the component TRACE section...
4 %}
```

The embedded C code is appended to the component `TRACE` section, and all its internal variables (as well as all the `DECLARE` instrument variables, *except* instrument parameters) may be used. To use instrument parameters, you should copy them into global variables in the DECLARE instrument section, and refer to these latter. This component declaration modifier is of course optional. You will find numerous usage examples, and in particular in the Sources section of the Component manual.

In some peculiar configurations it is necessary to position one or more groups of components, nested, in parallel, or overlapping. One example is a multiple crystal monochromator. One would then like the neutron ray to interact with *one of* the components of the group and then continue.

In order to handle such arrangements without removing neutrons, groups are defined by the `GROUP` modifier (after the AT-ROTATED positioning):

```
GROUP name
```

to all involved component declarations. All components of the same named group are tested one after the other, until one of them interacts (uses the SCATTER macro). The selected component acts on the neutron ray, and the rest of the group is skipped. Such groups are thus exclusive (only one of the elements is active).

Within a `GROUP`, *all* `EXTEND` sections of the group are executed. In order to discriminate components that are active from those that are skipped, one may use the SCATTERED flag, which is set to zero when entering each component or group, and incremented when the neutron is SCATTERed, as in the following example

```
COMPONENT name0 = comp (
                    p_1 = e_1,
                    p_2 = e_2,
                    ...)
AT (0,0,0) ABSOLUTE

COMPONENT name1 = comp...
AT (...) ROTATED (...)

GROUP GroupName EXTEND
%{
   if (SCATTERED) printf("I scatter"); else printf("I do not scatter");
%}

COMPONENT} name2 = comp ...
AT (...) ROTATED (...)
GROUP GroupName
```

Components *name1* and *name2* are at the same position. If the first one intercepts the neutron (and has a SCATTER within its `TRACE` section), the SCATTERED variable becomes true, the code extension will result in printing "I scatter", and the second component will be skipped. Thus, we recommend to make use of the SCATTER keyword each time a component 'uses' the neutron (scatters, detects, ...) within component definitions (see section 5.5). Also, the components to be grouped should be consecutive in the TRACE section of the instrument, and the GROUPed section should not contain components which are not part of the group.

A usage example of the GROUP keyword can be found in the `Neutron site/ILL/ILL_H15_IN6` instrument from the `mcgui`, to model 3 monochromators.

Combining EXTEND, GROUP and WHEN can result in unexpected behavior. Please read the related warning at the end of section 5.4.4.

5.4.3. Duplication of component instances - COPY

Often, one has a set of similar component instances in an instrument. These could be e.g. a set of identical monochromator blades, or a set of detectors or guide elements. Together with JUMPs (see below), there is a way to copy a component instance, duplicating parameter set.

Position (AT) and rotation (ROTATED) specification must be explicitly entered in order to avoid component overlapping, and it is futher recommended to explicitly set any EXTEND, GROUP, JUMP and WHEN keyword to each *mother* or copied instance. But **BEWARE**, advanced combinations of (many) keywords *may* introduce unexpected behaviour.

The syntax for instance copy is

```
COMPONENT name = COPY (instance_name)
```

where *instance_name* is the name of a preceding component instance in the instrument. It may be 'PREVIOUS' as well.

If you would like to change only some of the parameters in the instance copy, you may write, e.g.:

```
COMPONENT name = COPY (instance_name)(par1=0, par2=1)
```

which will override the original instance parameter values. This possibility to override parameters is very useful in case of describing e.g. sample environments using the Isotropic_Sqw and PowderN components, which allow *concentric* geometry (first instance must have `concentric = 1` and the second `concentric = 0`). In case EXTEND, GROUP, JUMP and WHEN keywords are defined for the copied instance, these will override the settings from the copied instance.

In the case where there are many duplicated components all originating from the same instance, there is a mechanism for automating copied instance names:

```
COMPONENT COPY(root_name) = COPY(instance_name)
```

will concatenate a unique number to *root_name*, avoiding name conflicts. As a side effect, referring to this component instance (for e.g. further positioning) is not straight forward as the name is determined by `mcstas` and does not depend completely on the user's choice, even though the PREVIOUS keyword may still be used. We thus recommend to use this naming mechanism only for components which should not be referred to in the instrument.

This automatic naming may be used anywhere in the TRACE section of the instrument, so that all components which do not need further referring may be labeled as COPY(Origin).

As an example, we show how to build a guide made of equivalent elements. Only the first instance of the Guide component is defined, whereas following instances are copies of that definition. The instance name of Guide components is set automatically.

```
COMPONENT CG_In = Arm() AT (...)

```

```
COMPONENT CG_1  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS

COMPONENT COPY(CG_1)  = COPY(CG_1)
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS

COMPONENT COPY(CG_1)  = COPY(CG_1)
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS
...
COMPONENT CG_Out = Arm() AT (0,0,L/n) RELATIVE PREVIOUS
```

5.4.4. Conditional components - WHEN

One of the most useful features of the extended McStas syntax is the conditional `WHEN` modifier. This optional keyword comes before the AT-ROTATED positioning. It basically enables the component only when a given condition is true (non null).

```
  COMPONENT name = comp (p_1 = e_1, p_2 = e_2, ...)
  WHEN condition
```

The condition has the same scope as the EXTEND modifier, i.e. may use component internal variables as well as all the `DECLARE` instrument variables.

Usage examples could be to have specific monitors only sensitive to selected processes, or to have components which are only present under given circumstances (e.g. removable guide or radial collimator), or to select a sample among a set of choices.

In the following example, an EXTEND block sets a condition when a scattering event is encountered, and the following monitor is then activated.

```
COMPONENT Sample = V_sample(...) AT ...
  EXTEND
  %{
    if (SCATTERED) flag=1; else flag=0;
  %}

COMPONENT MyMon = Monitor(...) WHEN (flag==1)
  AT ...
```

The WHEN keyword only applies to the TRACE section and related EXTEND blocks of instruments/components. Other sections (INITIALIZE, SAVE, MCDISPLAY, FINALLY) are executed independently of the condition. As a side effect, the 3D view of the instrument (mcdisplay) will show all components as if all conditions were true.

Also, the `WHEN` keyword is a condition for `GROUP`. This means that when the `WHEN` is false, the component instance is not active in the `GROUP` it belongs to.

A usage example of the WHEN keyword can be found in the `Neutron site/ILL/ILL_TOF_Env` instrument from the `mcgui`, to monitor neutrons depending on their fate.

WARNING: Combining WHEN, EXTEND and GROUP can result in unexpected behavior, please use with caution! Let for instance a GROUP of components all have the same WHEN condition, i.e. if the WHEN condition is false, none of the elements SCATTER, meaning that all neutrons will be ABSORBed. As a solution to this problem, we propose to include an EXTENDed Arm component in the GROUP, but with the opposite WHEN condition and a SCATTER keyword in the EXTEND section. This means that when none of the other GROUP elements are present, the Arm will be present and SCATTER.

5.4.5. Component loops and non sequential propagation - JUMP

There are situations for which one would like to repeat a given component many times, or under a given condition. The JUMP modifier is meant for that and should be placed after the positioning, GROUP and EXTEND. This breaks the sequential propagation along components in the instrument description. There may be more than one JUMP per component instance.

The jump may depend on a condition:

```
COMPONENT name = comp(p_1 = e_1, p_2 = e_2, ...)
  AT (...)
  JUMP reference WHEN condition
```

in which case the instrument TRACE will jump to the *reference* when *condition* is true.

The *reference* may be an instance name, as well as PREVIOUS, PREVIOUS(n), MYSELF, NEXT, and NEXT(n), where n is the index gap to the target either backward (PREVIOUS) or forward (NEXT), so that PREVIOUS(1) is PREVIOUS and NEXT(1) is NEXT. MYSELF means that the component will be iterated as long as the condition is true. This may be a way to handle multiple scattering, if the component has been designed for that.

The jump arrives directly inside the target component, in the local coordinate system (i.e. without applying the AT and ROTATED keywords). In order to control better the target positions, it is *required* that, except for looping MYSELF, the target component type should be an *Arm*.

There is a more general way to iterate components, which consists in repeating the loop for a given number of times.

```
  JUMP reference ITERATE number_of_times
```

This method is specially suited for very long curved guides of similar components, but in order to take into account rotation and translation between guide sections, the iterations are performed between Arm's.

In the following example for a curved guide made on $n = 500$ elements of length L on a curvature radius R, with gaps d between elements, we simply write:

```
COMPONENT CG_In = Arm() AT (...)

```

```
COMPONENT CG_1  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS

COMPONENT CG_2_Position = Arm()
  AT (0,0,L/n+d) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS

COMPONENT CG_2  = Guide_gravity(l=L/n, m=1, ...)
  AT (0,0,0) RELATIVE PREVIOUS
  ROTATED (0, (L/n+d)/R*180/PI, 0) RELATIVE PREVIOUS
  JUMP CG_2_Position ITERATE n
...
COMPONENT CG_Out = Arm() AT (0,0,L/n) RELATIVE PREVIOUS
```

Similarly to the `WHEN` modifier (see section 5.4.4), `JUMP` only applies within the TRACE section of the instrument definition. Other sections (INITIALIZE, SAVE, MCDISPLAY, FINALLY) are executed independently of the jump. As a side effect, the 3D view of the instrument (mcdisplay) will show components as if there was no jump. This means that in the following example, the very long guide 3D view only shows a single guide element.

It is *not* recommended to use the `JUMP` inside `GROUP`s, as the JUMP condition/counter applies to the component instance within its group.

We would like to emphasize the potential errors originating from such jumps. Indeed, imbricating many jumps may lead to situations were it is difficult to understand the flow of the simulation. We thus recommend the usage of JUMPs only for experienced and cautious users.

5.4.6. Enhancing statistics reaching components - SPLIT

The following method applies when the incoming neutron event distribution is considered to be representative of the real beam, but neutrons are lost in the course of propagation (with low efficiency processes, absorption, etc). Then, one may think that it's a pity to have so few events reaching the 'interesting' part of the instrument (usually close to the end of the instrument description). If some components make extensive use of random numbers (MC choices), they shuffle this way the distributions, so that identical incoming events will not produce the same outgoing event. In this case, you may use the `SPLIT` keyword with the syntax

```
  SPLIT r COMPONENT name = comp(...)
```

where the optional number r specifies the number of repetitions for each event. Default is $r = 10$. Each neutron event reaching component *name* will be repeated r times with a weight divided by r, so that in practice the number of events for the remaining part of the simulation (down to the END), will potentially have more statistics. This is only true if following components (and preferably component *name*) use random numbers. You may use this method as many times as you wish in the same instrument, e.g. at the monochromator and sample position. This keyword can also be used within a GROUP. The efficiency is roughly r raised to the number of occurrences in the instrument, so

that enhancing two components with the default $r = 10$ will produce at the end an enhancement effect of 100 in the number of events. The execution time will usually get slightly longer. This technique is known as the *stratified sampling* (see Appendix 3). If the instrument makes use of global variables - e.g. in conjunction with a WHEN or User Variable monitoring (see Monitor_nD) - you should take care that these variables are set properly for each SPLIT loop, which usually means that they must be reset inside the SPLITed section and assigned/used further on.

A usage example of the SPLIT keyword can be found in the `Neutron site/ILL/ILL_H15_IN6` instrument from the `mcgui`, to enhance statistics for neutrons scattering on monochromators and sample.

5.4.7. Attaching metadata to components and instruments - METADATA

The `METADATA` keyword attaches an arbitrary, named, typed block of text to a component *definition*, a component *instance*, or an instrument. It has no effect whatsoever on the simulated physics; it exists so that instrument/component authors can embed structured side-information (an IDF fragment, a JSON blob of geometry metadata, a CIF snippet, a free-form note, ...) that downstream tools can retrieve without having to parse the generated C code. This is how, e.g., `mcrun -format=NeXus -IDF` (section 4.4) locates instrument-definition-file fragments attached to detector components.

The syntax is

```
METADATA type name
%{
  ... arbitrary text ...
%}
```

placed after a component instance's positioning/EXTEND/JUMP clauses in the TRACE section, or inside a component definition (any number of `METADATA` blocks may be given). *type* and *name* are either bare identifiers or quoted strings (the latter allowing e.g. MIME-style types such as `"application/json"`); *name* identifies this particular metadata block within its owning component/instrument.

Metadata attached at the component-*definition* level is inherited by every instance of that component (and, per the usual `INHERIT` rules, by any component that `INHERIT`s from it); metadata attached to a specific *instance* in the TRACE section applies only to that instance, and instance-level METADATA with the same *name* overrides definition-level METADATA of that name for that instance.

Metadata blocks are inspected from the command line via `mcrun`: `-meta-list` prints all component:name pairs carrying metadata in the instrument, `-meta-defined=comp[:name]` indicates whether a given metadata name is defined (or lists all defined names for a component), `-meta-type=comp:name` prints its declared type, and `-meta-data=comp:name` prints its content – see Table 4.2 in the User Manual.

5.5. Writing component definitions

The purpose of a McStas component is to model the interaction of a neutron with a physical component of a real instrument. Given the state of the incoming neutron ray, the component definition calculates the state of the neutron ray when it leaves the component. The calculation of the effect of the component on the neutron is performed by a block of embedded C code. One example of a component definition is given in section 5.5.12, and all component definitions can be found on the McStas web-page [Mcsa] and are described in the McStas component manual.

There exists a large number of functions and constants available in order to write efficient components. See appendix B for

- neutron propagation functions
- geometric intersection time computations
- mathematical functions
- random number generation
- physical constants
- coordinate retrieval and operations
- file generation routines (for monitors),
- data file reading

5.5.1. The component definition header

```
1 DEFINE COMPONENT name
```

This marks the beginning of the definition, and defines the name of the component.

```
1 DEFINITION PARAMETERS (d_1, d_2, ...)
2 SETTING PARAMETERS (s_1, s_2, ...)
```

This declares the definition and setting parameters of the component. These parameters can be accessed from all sections of the component (see below), as well as in `EXTEND` sections of the instrument definition (see section 5.3).

Setting parameters are translated into C variables usually of type `double` in the generated simulation program, so they are usually numbers. Definition parameters are translated into `#define` macro definitions, and so can have any type, including strings, arrays, and function pointers.

Beyond the basic `double`/`int`/`string` (`char *`) types, two further parameter types are supported (both usable for `DEFINITION` and `SETTING` parameters):

- `vector` (equivalently `double *`): a parameter that is either a numeric array literal, e.g. `axis={0,1,0}`, or the name of a `double` array declared in the instrument `DECLARE` section. Vector parameters are optional by default (defaulting to `NULL`) unless an explicit default is given.

- `symbol`: accepts a bare C identifier as its value, useful for passing e.g. the name of a user-supplied function or a preprocessor symbol through to the component's C code.

However, because of the use of `#define`, definition parameters suffer from the usual problems with C macro definitions. Also, it is not possible to use a general C expression for the value of a definition parameter in the instrument definition, only constants and variable names may be used. For this reason, setting parameters should be used whenever possible.

Outside the `INITIALIZE` section of components, changing setting parameter values only affects the current section.

There are a few cases where the use of definition parameters instead of setting parameters makes sense. If the parameter is not numeric, nor a character string (*i.e.* an array, for example), a setting parameter cannot be used. Also, because of the use of `#define`, the C compiler can treat definition parameters as constants when the simulation is compiled. For example, if the array sizes of a multidetector are definition parameters, the arrays can be statically allocated in the component `DECLARE` section. If setting parameters were used, it would be necessary to allocate the arrays dynamically using *e.g.* `malloc()`.

Setting parameters may optionally be declared to be of type `int`, `char *` and `string`, just as in the instrument definition (see section 5.3).

```
1 OUTPUT PARAMETERS (s_1, s_2, ...)
```

This declares a list of C identifiers (variables, functions) that are output parameters (*i.e.* global) for the component. Output parameters are used to hold values that are computed by the component itself, rather than being passed as input. This could for example be a count of neutrons in a detector or a constant that is precomputed to speed up computation.

Using `OUTPUT PARAMETERS` is *strongly recommended* for `DECLARE` and internal/global component variables and functions in order to prevent that instances of the same component use the same variable names. Moreover (see section 5.5.4 below), these may be accessed from any other instrument part (e.g. using the `MC_GETPAR` C macro). On the other hand, the variables from the SHARE sections should *not* be defined as OUTPUT parameters.

The `OUTPUT PARAMETERS` section is optional.

Optional component parameters

Just as for instrument parameters, the definition and setting parameters of a component may be given a default value. Parameters with default values are called *optional parameters*, and need not be given an explicit value when the component is used in an

instrument definition. A parameter is given a default value using the syntax "*param* = *value*". For example

```
SETTING PARAMETERS (radius, height, pack= 1)
```

Here `pack` is an optional parameter and if no value is given explicitly, "1" will be used. In contrast, if no value is given for `radius` or `height`, an error message will result.

Optional parameters can greatly increase the convenience for users of components with many parameters that have natural default values which are seldom changed. Optional parameters are also useful to preserve backwards compatibility with old instrument definitions when a component is updated. New parameters can be added with default values that correspond to the old behavior, and existing instrument definitions can be used with the new component without changes.

However, optional parameters should not be used in cases where no general natural default value exists. For example, the length of a guide or the size of a slit should not be given default values. This would prevent the error messages that should be given in the common case of a user forgetting to set an important parameter.

5.5.2. The `NOACC` flag

A component *definition* may be marked

```
DEFINE COMPONENT name NOACC
```

to declare that this component class can never be compiled for GPU/OpenACC execution (section 4.5.4 in the User Manual) – e.g. because its `TRACE` section calls out to a C library that has no GPU-compatible implementation. This automatically pulls in the `-DFUNNEL` dependency flag, which enables McStas's CPU/GPU "funneling" mode so that an instrument mixing GPU-friendly and `NOACC` components can still be run with `--openacc`, at the cost of funneling neutron rays that reach a `NOACC` component back to the CPU for that component's computation. The complementary, per-*instance* keyword `CPU` (see section 5.4) forces one specific instance of an otherwise GPU-capable component to run on CPU only, e.g. for debugging.

5.5.3. The `DEPENDENCY` line

```
DEPENDENCY "-lLIB1 -lLIB2 .."
```

This optional line indicates any external library which are needed by the component. It should be used when the component code calls external functions and routines which should be available when compiling the instrument. This line must be indicated just before the `SHARE` and `DECLARE` sections. Refer to the section 5.3.2 for more information.

5.5.4. The `DECLARE` section

```
DECLARE
%{
// C code declarations (variables, definitions, functions)
// These are usually OUTPUT parameters to avoid name conflicts
%}
```

This gives C declarations of global variables, functions, etc. that are used by the component code. This may for instance be used to declare a neutron counter for a detector component. This section is optional.

Note that any variables declared in a `DECLARE` section are *global*. Thus a name conflict may occur if two instances of a component are used in the same instrument. To avoid this, variables declared in the `DECLARE` section should be `OUTPUT` parameters of the component because McStas will then rename variables to avoid conflicts. For example, a simple detector might be defined as follows:

```
DEFINE COMPONENT Detector
OUTPUT PARAMETERS (counts)
DECLARE
%{
  int counts;
%}
...
```

The idea is that the `counts` variable counts the number of neutrons detected. In the instrument definition, the `counts` parameter may be referenced using the `MC_GETPAR` C macro, as in the following example instrument fragment:

```
COMPONENT d1 = Detector()
...
COMPONENT d2 = Detector()
...
FINALLY
%{
  printf("Detector counts: d1 = %d, d2 = %d\n",
         MC_GETPAR(d1,counts), MC_GETPAR(d2,counts));
%}
```

This way, McStas takes care to rename transparently the two 'counts' `OUTPUT` parameters so that they are distinct, and can be accessed from elsewhere in the instrument (EXTEND, FINALLY, SAVE, ...) or from other components. This particular example is outdated since McStas monitors will themselves output their contents.

5.5.5. The `SHARE` section

```
SHARE
%{
// C code shared declarations (variables, definitions, functions)
// These should not be OUTPUT parameters
%}
```

The SHARE section has the same role as DECLARE except that when using more than one instance of the component, it is inserted *only once* in the simulation code. No occurrence of the items to be shared should be in the OUTPUT parameter list (not to have McStas rename the identifiers). This is particularly useful when using many instances of the same component (for instance guide elements). If the declarations were in the DECLARE section, McStas would duplicate it for each instance (making the simulation code longer). A typical example is to have shared variables, functions, type and structure definitions that may be used from the component TRACE section. For an example of SHARE, see the samples/Single_crystal component. The %include "file" keyword may be used to import a shared library. The SHARE section is optional and should be located *before* the DECLARE section.

5.5.6. The INITIALIZE section

```
1 INITIALIZE
2 %{
3 // C code initialization
4 %}
```

This gives C code that will be executed once at the start of the simulation, usually to initialize any variables declared in the DECLARE section. This section is optional. Component setting parameters may be modified in this section, affecting the rest of the component.

5.5.7. The TRACE section

```
1 TRACE
2 %{
3 // C code to compute neutron interaction with component
4 %}
```

This performs the actual computation of the interaction between the neutron ray and the component. The C code should perform the appropriate calculations and assign the resulting new neutron state to the state parameters. Most components will require propagation routines to reach the component entrance/area. Special macros PROP_Z0; and PROP_DT(); are provided to automate this process (see section B.1).

The C code may also execute the special macro ABSORB to indicate that the neutron has been absorbed in the component and the simulation of that neutron will be aborted. On the other hand, if the neutron event should be *allowed* be backpropagated, the special macro ALLOW_BACKPROP; should precede the call to the PROP_ call inside the component. When the neutron state is changed or detected, for instance if the component simulates multiple events as multiple reflections in a guide, the special macro SCATTER should be called. This does not affect the results of the simulation in any way, but it allows the front-end programs to visualize the scattering events properly, and to handle component GROUPs in an instrument definition (see section 5.3.9). It basically increments the SCATTERED

counter. The `SCATTER` macro should be called with the state parameters set to the proper values for the scattering event, so that neutron events are displayed correctly. For an example of `SCATTER`, see the optics/Guide component.

5.5.8. The `SAVE` section

```
SAVE
%{
// C code to execute in order to save data
%}
```

This gives code that will be executed when the simulation ends, or is requested to save data, for instance when receiving a USR2 signal (on Unix systems, see section 4.4), or when triggered by the `Progress_bar(flag_save=1)` component. This might be used by monitors and detectors in order to write results. An extension depending on the selected output format (see section 4.4) is automatically appended to file names, if these latter do not contain extension.

In order to work properly with the common output file format used in McStas, all monitor/detector components should use standard macros for writing data in the SAVE or FINALLY section, as explained below. In the following, we use $N = \sum_i p_i^0$ to denote the count of detected neutron events, $p = \sum_i p_i$ to denote the sum of the weights of detected neutrons, and $p2 = \sum_i p_i^2$ to denote the sum of the squares of the weights, as explained in section 3.2.1.

As a default, all monitors using the standard macros will display the integral p of the monitor bins, as well as the 2^{nd} moment σ and the number of statistical events N. This will result in a line such as:

```
Detector: CompName_I=p CompName_ERR=sigma CompNameN=N "filename"
```

For 1D, 2D and 3D monitors/detectors, the data histogram store in the files is given *p*er bin when the signal is the neutron intensity (*i.e.* most of the cases). Most monitors define binning for an x_n axis value as the sum of events falling into the $[x_n x_{n+1}]$ range, *i.e* the bins are *n*ot centered, but left aligned. Using the Monitor_nD component, it is possible to monitor other signals using the '`signal=`*variable_name*' in the 'options' parameter (refer to that component documentation).

Single detectors/monitors The results of a single detector/monitor are written using the following macro:

```
DETECTOR_OUT_0D(t, N, p, p2)
```

Here, t is a string giving a short descriptive title for the results, *e.g.* "Single monitor".

One-dimensional detectors/monitors The results of a one-dimensional detector/monitor are written using the following macro:

```
DETECTOR_OUT_1D(t,
          xlabel, ylabel,
          xvar, x_min, x_max, m,
          &N[0], &p[0], &p2[0],
         filename)
```

Here,

- t is a string giving a descriptive title (*e.g.* "Energy monitor"),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (*e.g.* "Energy [meV]"),
- *ylabel* is a string giving a descriptive label for the Y axis of a plot (*e.g.* "Intensity"),
- *xvar* is a string giving the name of the variable on the X axis (*e.g.* "E"),
- $x_{\min}$ is the lower limit for the X axis,
- $x_{\max}$ is the upper limit for the X axis,
- m is the number of elements in the detector arrays,
- &N[0] is a pointer to the first element in the array of N values for the detector component (or NULL, in which case no error bars will be computed),
- &p[0] is a pointer to the first element in the array of p values for the detector component,
- &*p2*[0] is a pointer to the first element in the array of *p2* values for the detector component (or NULL, in which case no error bars will be computed),
- *filename* is a string giving the name of the file in which to store the data.

Two-dimensional detectors/monitors The results of a two-dimensional detector/monitor are written to a file using the following macro:

```
DETECTOR_OUT_2D(t,
          xlabel, ylabel,
          x_min, x_max, y_min, y_max, m, n,
          &N[0][0], &p[0][0], &p2[0][0],
         filename)
```

Here,

- t is a string giving a descriptive title (*e.g.* "PSD monitor"),
- *xlabel* is a string giving a descriptive label for the X axis in a plot (*e.g.* "X position [cm]"),

- *ylabel* is a string giving a descriptive label for the Y axis of a plot (*e.g.* "Y position [cm]"),
- $x_{\min}$ is the lower limit for the X axis,
- $x_{\max}$ is the upper limit for the X axis,
- $y_{\min}$ is the lower limit for the Y axis,
- $y_{\max}$ is the upper limit for the Y axis,
- m is the number of elements in the detector arrays along the X axis,
- n is the number of elements in the detector arrays along the Y axis,
- &N[0][0] is a pointer to the first element in the array of N values for the detector component,
- &p[0][0] is a pointer to the first element in the array of p values for the detector component,
- &*p2*[0][0] is a pointer to the first element in the array of *p2* values for the detector component,
- *filename* is a string giving the name of the file in which to store the data.

Note that for a two-dimensional detector array, the first dimension is along the X axis and the second dimension is along the Y axis. This means that element (i_x, i_y) can be obtained as $p[i_x * n + i_y]$ if p is a pointer to the first element.

Three-dimensional detectors/monitors There is no three-dimensional DETECTOR_OUT macro. Three-dimensional data can be recorded in NeXus format with `Monitor_nD` in event (list) mode, using pixel identifiers.

5.5.9. The `FINALLY` section

```
1 FINALLY
2 %{
3 // C code to execute at end of simulation ...
4 %}
```

This gives code that will be executed when the simulation has ended. This might be used to free memory and print out final results from components, *e.g.* the simulated intensity in a detector. This section also triggers the SAVE section to be executed.

5.5.10. The `MCDISPLAY` section

```
MCDISPLAY
%{
// C code to draw a sketch of the component ...
%}
```

This gives C code that draws a sketch of the component in the plots produced by the `mcdisplay` front-end (see section 4.5.5). The section can contain arbitrary C code and may refer to the parameters of the component, but usually it will consist of a short sequence of the special commands described below that are available only in the MCDISPLAY section. When drawing components, all distances and positions are in meters and specified in the local coordinate system of the component.

The MCDISPLAY section is optional. If it is omitted, `mcdisplay` will use a default symbol (a small circle) for drawing the component.

The `magnify` command This command, if present, must be the first in the section. It takes a single argument: a string containing zero or more of the letters "x", "y" and "z". It causes the drawing to be enlarged along the specified axis in case `mcdisplay` is called with the `--zoom` option. For example:

```
    magnify("xy");
```

The `line` command The `line` command takes the following form:

```
line(x_1, y_1, z_1, x_2, y_2, z_2)
```

It draws a line between the points (x_1, y_1, z_1) and (x_2, y_2, z_2).

The `dashed_line` command The `dashed_line` command takes the following form:

```
dashed_line(x_1, y_1, z_1, x_2, y_2, z_2, n)
```

It draws a dashed line between the points (x_1, y_1, z_1) and (x_2, y_2, z_2) with n equidistant spaces.

The `multiline` command The `multiline` command takes the following form:

```
  \texttt{multiline(n, x_1, y_1, z_1, ..., x_n, y_n, z_n)}
```

It draws a series of lines through the n points (x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n). It thus accepts a variable number of arguments depending on the value of n. This exposes one of the nasty quirks of C since *no* type checking is performed by the C compiler. It is thus very important that all arguments to `multiline` (except n) are valid numbers of type `double`. A common mistake is to write

```
    multiline(3, x, y, 0, ...)
```

which will silently produce garbage output. This must instead be written as

```
1     multiline(3, (double)x, (double)y, 0.0, ...)
```

The `rectangle` command The `rectangle` command takes the following form:

```
1 rectangle(plane, x, y, z, width, height)
```

Here *plane* should be either `"xy"`, `"xz"`, or `"yz"`. The command draws a rectangle in the specified plane with the center at (x, y, z) and the size *width* $\times$ *height*. Depending on *plane* the width and height are defined as:

plane	*width*	*height*
xy	x	y
xz	x	z
yz	y	z

The `box` command The `box` command takes the following form:

```
1 box(x, y, z, xwidth, yheight, zlength)
```

The command draws a box with the center at (x, y, z) and the size *xwidth* $\times$ *yheight* $\times$ *zlength*.

The `circle` command The `circle` command takes the following form:

```
1 circle(plane, x, y, z, r)
```

Here *plane* should be either `"xy"`, `"xz"`, or `"yz"`. The command draws a circle in the specified plane with the center at (x, y, z) and the radius r.

5.5.11. The end of the component definition

```
1 END
```

This marks the end of the component definition.

5.5.12. A component example: Slit

A simple example of the component `Slit` is given.

5.6. Extending component definitions

Suppose you are interested by one component of the McStas library, but you would like to customize it a little. There are different ways to extend an existing component.

5.6.1. Extending from the instrument definition

If you only want to *add* something on top of the component existing behavior, the simplest is to work from the instrument definition `TRACE` section, using the `EXTEND` modifier (see section 5.4.2). You do not need to write a new component definition, but only add a piece of code to execute.

5.6.2. Component heritage and duplication

There is a heritage mechanism to create children of existing components. These are exact duplicates of the parent component, but one may override/extend original definitions of any section.

The syntax for a full component child is

```
DEFINE COMPONENT child_name INHERIT parent_name
```

This single line will copy all parts of the *parent* into the *child*, except for the documentation header.

As for normal component definitions, you may add other parameters, `DECLARE`, `TRACE`, ... sections. Each of them will replace or extend (be concatenated to, with the `INHERIT`/`EXTEND` keywords, see example below) the corresponding *parent* definition. In practice, you could inherit a component and only rewrite some of it, as in the following example:

```
DEFINE COMPONENT child_name INHERIT parent_name

  SETTING PARAMETERS (newpar1, newpar2)
  INITIALISE
  INHERIT parent_name
  EXTEND
  %{
// C code to be concatenated to the parent_name INITIALIZE
  %}
  SAVE
  %{
// C code to replace the parent_name SAVE
  %}
END
```

where two additional parameters have been defined, and should be handled in the extension of the original INITIALIZE section. Note that each per-section `INHERIT parent_name` (optionally followed by `EXTEND %{...%}`) is specified *inside* that section's own block, immediately after the section keyword, rather than as a separate top-level `COPY` clause.

A special care should be taken in mixing bits from different sources, specially concerning variables. This may result in difficulties to compile components.

On the other hand, if you do not want to derive a component as a whole from a parent (i.e. the top-level `DEFINE COMPONENT` line does not itself use `INHERIT`), you may still pull in specific sections from any component using the same per-section `INHERIT` syntax:

```
1  DEFINE COMPONENT name (...)
2  SETTING PARAMETERS (...)
3  DECLARE
4  INHERIT parent1
5  INITIALISE
6  INHERIT parent2
7  EXTEND
8  %{
9  // C code to be concatenated to the  parent2 INITIALIZE
10 %}
11 TRACE
12 INHERIT parent3
13 END
```

This mechanism may lighten the component code, but as above, mixing bits from different sources requires special care regarding variable naming.

5.7. McDoc, the McStas library documentation tool

McStas includes a facility called McDoc to help maintain good documentation of components and instruments. In the source code, comments may be written that follow a particular format understood by McDoc. The McDoc facility will read these comments and automatically produce output documentation in various forms. By using the source code itself as the source of documentation, the documentation is much more likely to be a faithful and up-to-date description of how the component/instrument actually works.

5.7.1. Documentation generators mcdoc and mcgui

Two forms of documentation can be generated. One is the component entry dialog in the `mcgui` front-end, see section 4.5.1. The other is a collection of web pages documenting the components and instruments, handled via the `mcdoc` front-end (see section 4.5.8), and the complete documentation for all available McStas components and instruments may be found at the McStas web page [Mcsa], as well as in the McStas library (see 9.1). All available McStas documentation is accessible from the `mcgui` 'Help' menu.

Note that McDoc-compliant comments in the source code are no substitute for a good reference manual entry. The mathematical equations describing the physics and algorithms of the component should still be written up carefully for inclusion in the component manual. The McDoc comments are useful for describing the general behavior of the component, the meaning and units of the input parameters, etc.

5.7.2. The format of the comments in the library source code

The format of the comments understood by McDoc is mostly straight-forward, and is designed to be easily readable both by humans and by automatic tools. McDoc has been written to be quite tolerant in terms of how the comments may be formatted and broken across lines. A good way to get a feeling for the format is to study some of the

examples in the existing components and instruments. Below, a few notes are listed on the requirements for the comment headers:

The comment syntax uses %IDENTIFICATION, %DESCRIPTION, %PARAMETERS, %EXAMPLE:, %LINKS, and %END keywords to mark different sections of the documentation. Keywords may be abbreviated (except for %EXAMPLE:), *e.g.* as %IDENT or %I.

Additionally, optional keys %VALIDATION and %BUGS may be found to list validation status and possible bugs in the component.

- In the %IDENTIFICATION section, author: (or written by: for backwards compatibility with old comments) denote author; date:, version:, and origin: are also supported. Any number of Modified by: entries may be used to give the revision history. The author:, date:, etc. entries must all appear on a single line of their own. Everything else in the identification section is part of a "short description" of the component.

- In the %PARAMETERS section, descriptions have the form "*name*: [*unit*] *text*" or "*name*: *text* [*unit*]". These may span multiple lines, but subsequent lines must be indented by at least four spaces. Note that square brackets [] should be used for units. Normal parentheses are also supported for backwards compatibility, but nested parentheses do not work well.

- The %DESCRIPTION section contains text in free format. The text may contain HTML tags like <IMG> (to include pictures) and <A>...</A> (for links to other web pages, but see also the %LINK section). In the generated web documentation pages, the text is set in <PRE>...</PRE>, so that the line breaks in the source will be obeyed.

- The %EXAMPLE: (or %TEST:) lines in instrument headers indicate an example parameter set or command that may be run to test the instrument. A %TESTSCAN: (older name %SCAN:) line defines a parameter scan test with one expected value per scan point. A following Detector: <name>_I=<value> indicates what value should be obtained for a given monitor. More than one example line may be specified in instruments.

- Any number of %LINK sections may be given; each one contains HTML code that will be put in a list item in the link section of the description web page. This usually consists of an <A HREF="..."> ... </A> pointer to some other source of information.

- Optionally, an %INSTRUMENT_SITE section followed by a single word is used to sort *instruments* by origin/location in the 'Neutron Site' menu in mcgui.

- After %END, no more comment text is read by McDoc.

6. Polarisation in McStas

McStas provides a set of components for simulating neutron beam polarisation and the optical elements that manipulate it: polarising monochromators, mirrors and guides, magnetic-field regions (including spin flippers and spin-echo coils), and polarisation-sensitive monitors. This chapter originated as an appendix written by P. Christiansen documenting the first polarisation components (2006); it is now promoted to a full chapter and brought up to date with the current McStas 3.x component set and the `pol-lib` field/precession framework that underlies it.

We first describe the polarisation vector and how it is related to the neutron wavefunction (section 6.1) and then the physics of the scattering processes relevant to polarisation (section 6.2). Section 6.3 describes in detail how McStas propagates the polarisation vector through magnetic fields – the numerical precession algorithm and the field functions available – since this underlies every magnetic-field component in the library. Section 6.4 then documents the current polarisation components themselves, organised by role, and section 6.5 lists example and test instruments, including a worked spin-echo example.

We rely heavily on the books [Lov84; Wil88] for the underlying physics, where the detailed calculations can be found.

The notation used here (and in [Wil88]) is P (scalar), $\mathbf{P}$ (vector), $\hat{\mathbf{P}}$ (unit-vector), $\hat{\sigma}$ (operator), and $\hat{\boldsymbol{\sigma}}$ (vector of operators).

6.1. The Polarization Vector

The spin of the neutron is represented by an operator $\hat{\mathbf{s}}$ for which only a single component can be measured at one time. Each single measurement will give a value $\pm 1/2$, but if we could make a large number of measurements on the same neutron state, in each of the three axis directions, and then make the average we get $\langle \hat{\mathbf{s}} \rangle$. The polarization vector, $\mathbf{P}$, is then defined as:

$$\mathbf{P} = \frac{\langle \hat{\mathbf{s}} \rangle}{s}, \tag{6.1}$$

so that $-1 \leq |\mathbf{P}| \leq +1$

For a neutron beam which contains N neutrons, each with a polarization $\mathbf{P}_i$, the beam polarization is defined as:

$$\mathbf{P} = \frac{\sum_i \mathbf{P}_i}{N}. \tag{6.2}$$

If we have one common quantization direction (e.g. a magnetic field direction) each neutron will either be spin up, $\uparrow$, or spin down, $\downarrow$, and the polarization can be expressed as:

$$P = \frac{n^{\uparrow} - n^{\downarrow}}{n^{\uparrow} + n^{\downarrow}}, \tag{6.3}$$

where ν ($n^{\downarrow}$) is the number of neutrons with spin up (down).

For a given neutron the probability of the neutron being spin up, $P^{\uparrow}$, is:

$$P^{\uparrow} = \frac{n^{\uparrow}}{n^{\uparrow} + n^{\downarrow}} = \frac{n^{\uparrow} + (n^{\downarrow} - n^{\downarrow})/2}{n^{\uparrow} + n^{\downarrow}} = \frac{1+P}{2}, \tag{6.4}$$

and $P^{\downarrow} = 1 - P^{\uparrow} = (1-P)/2$.

The expectation value of the 'spin' operator, $\hat{\boldsymbol{\sigma}}$, which can be expressed by the Pauli matrices, is the polarization vector $\mathbf{P}$, $\mathbf{P} = \langle\hat{\boldsymbol{\sigma}}\rangle \equiv \langle\chi|\hat{\boldsymbol{\sigma}}|\chi\rangle$. The most general form of the spin wave-function χ for a neutron (spin 1/2) is:

$$\chi = a\chi_{\uparrow} + b\chi_{\downarrow}, \tag{6.5}$$

where $\chi_{\uparrow}$ and $\chi_{\downarrow}$ are eigenfunction of $\hat{\sigma}^z$, and the complex coefficients a and b satisfy $|a|^2 + |b|^2 = 1$.

By calculation we find:

$$P_x = \langle\chi|\hat{\sigma}_x|\chi\rangle = 2\mathrm{Re}(a^*b) \tag{6.6}$$

$$P_y = \langle\chi|\hat{\sigma}_y|\chi\rangle = 2\mathrm{Im}(a^*b) \tag{6.7}$$

$$P_z = \langle\chi|\hat{\sigma}_z|\chi\rangle = |a|^2 - |b|^2 \tag{6.8}$$

This shows the relation of the polarization vector to the neutron wave function.

The neutron magnetic moment operator can be expressed in terms of $\hat{\boldsymbol{\sigma}}$, as:

$$\hat{\boldsymbol{\mu}} = \mu_n\hat{\boldsymbol{\sigma}}, \tag{6.9}$$

which, as shown above, is related to the polarization vector.

In our simulation we represent the polarization by the vector $\mathbf{S} = (s_x, s_y, s_z)$ which is propagated through the different components so it has the correct relative orientation in each component. The probability for the spin to be parallel a given direction $\mathbf{n}$ is then:

$$P(\uparrow|\mathbf{n}) = \frac{1 + \mathbf{n}\cdot\mathbf{S}}{2}. \tag{6.10}$$

This equation (from [SD01]) is easy to understand. The average spin along $\mathbf{n}$ is $\mathbf{n}\cdot\mathbf{S}$ and the probability then follows from Eq. 6.4.

For an unpolarized beam, $\mathbf{S} = \mathbf{0}$ and all directions are equally probable (50 %).

Note that in our approach we do not decide if the neutron is up or down after a given component, but instead keep track of as much information for as long as possible.

In the following we will use $\mathbf{P}$ to denote the polarization vector. The most important variables used are:

$\mathbf{Q}$	Scattering vector.
$\mathbf{P}$	Polarization before a component (ingoing).
$\mathbf{P}_\perp$	Polarization perpendicular to scattering vector, $\mathbf{P}_\perp = \hat{\mathbf{Q}} \times (\mathbf{P} \times \hat{\mathbf{Q}})$.
$\mathbf{P}'$	Polarization after a component (outgoing).
$\hat{\mathbf{N}}$	Unit vector in direction of atomic spin ($\hat{\mathbf{N}} \cdot \tilde{\mathbf{B}} = -1$ for a ferromagnet).
F_N	Unit cell nuclear structure factor.
F_M	Unit cell magnetic structure factor.

The unit cell nuclear structure factor is defined as:

$$F_N = \sum_{\mathbf{d}} \exp(i\mathbf{Q} \cdot \mathbf{d}) \bar{b}_d, \tag{6.11}$$

where the $\mathbf{d}$ is the position of the d'th atom within the unit cell, and $\bar{b}_d$ is the average of b_d. In the simple case of a single atom Bravais crystal one finds $F_N = \bar{b}$.

The unit cell magnetic structure factor is useful when the atoms in the crystal only have spin orbital angular momentum, and simple when the magnet is saturated (all spins are parallel or anti-parallel to *one* direction, $\sigma_d = \pm 1$). It is then given as:

$$F_M = \gamma_n r_0 \sum_{\mathbf{d}} \exp(i\mathbf{Q} \cdot \mathbf{d}) \frac{1}{2} g_d F_d(\mathbf{Q}) \langle \hat{S}_d \rangle \sigma_d, \tag{6.12}$$

where $r_0 = \frac{\mu_0}{4\pi} \frac{e^2}{m_e} = 2.818 \times 10^{-15}$m, $g = 2$ is the Landé splitting factor, and $F_d(\mathbf{Q})$ is the magnetic form factor, which is the Fourier transform of the magnetization density (normalized so that $F_d(0) = 1$), and $\langle \hat{S}_d \rangle$ is the thermal average of the ordered atomic spin.

In the following the Debye-Weller factor ($\exp(-W_d)$) have been ignored in all cross sections.

6.1.1. Example: Magnetic fields

The magnetic moment operator of the neutron is $\hat{\boldsymbol{\mu}} = \gamma_n \hat{\mathbf{s}}$, where $\gamma_n = 2\mu_n = -3.826$ is the gyromagnetic ratio (spin and magnetic moment is anti-parallel as for an electron) [1]

A magnetic field, $\mathbf{B}$, will exert a torque, $\boldsymbol{\tau} = d\mathbf{s}/dt = (1/\gamma_n) d\boldsymbol{\mu}/dt$, on the neutron magnetic moment:

$$\frac{1}{\gamma_n} \frac{d\boldsymbol{\mu}}{dt} = \boldsymbol{\mu} \times \mathbf{B} \tag{6.13}$$

The magnetic moment μ can be related to the polarization as $\boldsymbol{\mu} = \gamma_n \mathbf{P}/2$, and inserting in Eq. 6.13 we find:

$$\frac{d\mathbf{P}}{dt} = \gamma_n \mathbf{P} \times \mathbf{B} \tag{6.14}$$

In the simple case where $\mathbf{B} = (0, 0, B)$, we find the solution ([Wil88] p. 18) :

[1] Note that if we had used S (with values $S = \pm 1$) to define γ_n we would get $\gamma_n = -1.913$ which is also commonly used.

$$
\begin{aligned}
P_X(t) &= \cos(\omega_L t) P_X(0) - \sin(\omega_L t) P_Y(0) \\
P_Y(t) &= \sin(\omega_L t) P_X(0) + \cos(\omega_L t) P_Y(0) \\
P_Z(t) &= P_Z(0),
\end{aligned}
\tag{6.15}
$$

where $\omega_L = -\gamma_n B/\hbar$ is the Larmor frequency.

> The equations above were checked against the equations in the "polarimetrie neutronique" notes by Francis Tasset and found to be consistent. There can be sign differences between different publications depending on whether they use a right-handed (like e.g. McStas) or a left-handed (like e.g. NISP) coordinate system.

This closed-form solution is exactly what the **Pol_constBfield** and **Pol_FieldBox** components (section 6.4) evaluate directly in one step for a spatially uniform field. For the general case of a spatially varying, rotating, or otherwise non-uniform field, McStas instead integrates the precession numerically – this general algorithm is described in section 6.3, and is exactly the same physics applied piecewise over short enough intervals that the field can be treated as locally uniform.

6.2. Polarized Neutron Scattering

First we will give a short introduction to how calculations are done and then quote some results which are important for implementing the first McStas components.

All the potentials (nuclear, magnetic, and electric) we will be interested in can be written on the form:

$$
\hat{v} = \boldsymbol{\beta} + \boldsymbol{\alpha} \cdot \hat{\boldsymbol{\sigma}}
\tag{6.16}
$$

The first term does not affect the spin, while the second term can change the spin. Let us just remind here that:

$$
\begin{aligned}
&\hat{\sigma}_x \chi_\uparrow = \chi_\downarrow, \quad \hat{\sigma}_y \chi_\uparrow = i\chi_\downarrow, \quad \hat{\sigma}_z \chi_\uparrow = \chi_\uparrow, \\
&\hat{\sigma}_x \chi_\downarrow = \chi_\uparrow, \quad \hat{\sigma}_y \chi_\downarrow = -i\chi_\uparrow, \quad \hat{\sigma}_z \chi_\downarrow = -\chi_\downarrow.
\end{aligned}
\tag{6.17}
$$

So that the interaction proportional to $\hat{\sigma}_x$ and $\hat{\sigma}_y$ results in spin flips, while the interactions with $\hat{\sigma}_z$ conserves the spin.

It turns out to be smart to define a density matrix operator:

$$
\hat{\rho} = \chi\chi^\dagger = \begin{pmatrix} |a|^2 & ab^* \\ ba^* & |b|^2 \end{pmatrix} = \frac{1}{2}(\mathcal{I} + \mathbf{P} \cdot \hat{\boldsymbol{\sigma}}),
\tag{6.18}
$$

where χ is the neutron wave function (Eq. 6.5), and $\mathcal{I}$ is the unit matrix.

Using the density matrix the elastic cross section can be written as ([Lov84], Eq. 10.31):

$$
\frac{d\sigma}{d\Omega} = \mathrm{Tr}\hat{\rho}\hat{v}^\dagger\hat{v} = \sum_{\lambda,\lambda'} p_\lambda \mathrm{Tr}\hat{\rho}\langle\lambda|\hat{V}^\dagger(\mathbf{Q})|\lambda'\rangle\langle\lambda'|\hat{V}(\mathbf{Q})|\lambda\rangle\delta(E_\lambda - E_{\lambda'}),
\tag{6.19}
$$

where $\hat{V}$ is the interaction potential and it is understood that the trace is to be taken with respect only to the neutron spin coordinates. The outgoing polarization is given as:

$$\mathbf{P}'\frac{d\sigma}{d\Omega} = \mathrm{Tr}\hat{\rho}\hat{v}^\dagger\hat{\boldsymbol{\sigma}}\hat{v} = \sum_{\lambda,\lambda'} p_\lambda \mathrm{Tr}\hat{\rho}\langle\lambda|\hat{V}^\dagger(\mathbf{Q})|\lambda'\rangle\hat{\boldsymbol{\sigma}}\langle\lambda'|\hat{V}(\mathbf{Q})|\lambda\rangle\delta(E_\lambda - E_{\lambda'}) \tag{6.20}$$

Inserting Eq. 6.16 in Eq. 6.19 and Eq. 6.20 results in the two master equations for polarized neutron scattering:

$$\mathrm{Tr}\hat{\rho}\hat{v}^\dagger\hat{v} = \boldsymbol{\alpha}^\dagger\cdot\boldsymbol{\alpha} + \boldsymbol{\beta}^\dagger\boldsymbol{\beta} + \boldsymbol{\beta}^\dagger(\boldsymbol{\alpha}\cdot\mathbf{P}) + (\boldsymbol{\alpha}^\dagger\cdot\mathbf{P})\boldsymbol{\beta} + i\mathbf{P}\cdot(\boldsymbol{\alpha}^\dagger\times\boldsymbol{\alpha}), \tag{6.21}$$

and

$$\mathrm{Tr}\hat{\rho}\hat{v}^\dagger\hat{\boldsymbol{\sigma}}\hat{v} = \boldsymbol{\beta}^\dagger\boldsymbol{\alpha}+\boldsymbol{\alpha}^\dagger\boldsymbol{\beta}+\boldsymbol{\beta}^\dagger\boldsymbol{\beta}\mathbf{P}+\boldsymbol{\alpha}^\dagger(\boldsymbol{\alpha}\cdot\mathbf{P})+(\boldsymbol{\alpha}^\dagger\cdot\mathbf{P})\boldsymbol{\alpha}-\mathbf{P}(\boldsymbol{\alpha}^\dagger\cdot\boldsymbol{\alpha})-i\boldsymbol{\alpha}^\dagger\times\boldsymbol{\alpha}+i\boldsymbol{\beta}^\dagger(\boldsymbol{\alpha}\times\mathbf{P})+i(\mathbf{P}\times\boldsymbol{\alpha}^\dagger)\boldsymbol{\beta}. \tag{6.22}$$

Based on these two equations and the interaction potentials all the results presented in the following are derived in [Lov84].

6.2.1. Example: Nuclear scattering

The nuclear scattering potential for a crystal is:

$$\hat{V}_N(\mathbf{Q}) = \sum_{\mathbf{l},\mathbf{d}} \exp(i\mathbf{Q}\cdot\mathbf{R}_{ld})(A_{ld} + \frac{1}{2}B_{ld}\hat{\boldsymbol{\sigma}}\cdot\hat{\mathbf{I}}_{ld}), \tag{6.23}$$

so that

$$\begin{aligned}\boldsymbol{\alpha} &= \sum_{\mathbf{l},\mathbf{d}} \exp(i\mathbf{Q}\cdot\mathbf{R}_{ld})\frac{1}{2}B_{ld}\hat{\mathbf{I}}_{ld}\\ \boldsymbol{\beta} &= \sum_{\mathbf{l},\mathbf{d}} \exp(i\mathbf{Q}\cdot\mathbf{R}_{ld})A_{ld},\end{aligned} \tag{6.24}$$

where $\hat{\mathbf{I}}$ is the nuclear spin operator and the constants A and B are related to the nuclear scattering lengths b^+ and b^- as $A = ((I+1)b^+ + Ib^-)/(2I+1)$ and $B = (b^+ + b^-)/(2I+1)$.

To calculate the polarization cross section and outgoing polarization we have to average over the nuclear spin (which we assume is random oriented), so that terms linear in $\boldsymbol{\alpha}$ (three last terms in Eq. 6.21) disappears. The scattering cross section ends up being (see [Lov84] p. 159):

$$\frac{d\sigma}{d\Omega} = \sum_{\mathbf{l},\mathbf{d},\mathbf{l}',\mathbf{d}'} \exp(i\mathbf{Q}\cdot(\mathbf{R}_{ld}-\mathbf{R}_{l'd'}))(\overline{|F_N(\mathbf{Q})|^2} + \delta_{\mathbf{l},\mathbf{l}'}\delta_{\mathbf{d},\mathbf{d}'}[\left|\overline{F_N(\mathbf{Q})}\right|^2 - \overline{|F_N(\mathbf{Q})|^2} + \frac{1}{4}\overline{|B_{ld}|^2}]) \tag{6.25}$$

where the first term is the coherent cross-section and the second term is the site-incoherent cross-section. Both terms are independent of $\mathbf{P}$ as expected for a system without a preferred internal direction.

The polarization in the final state is:

$$\mathbf{P}'\frac{d\sigma}{d\Omega} = \sum_{\mathbf{l},\mathbf{d},\mathbf{l}',\mathbf{d}'} \exp(i\mathbf{Q}\cdot(\mathbf{R}_{ld}-\mathbf{R}_{l'd'}))\mathbf{P}'(\overline{|F_N(\mathbf{Q})|^2}+\delta_{\mathbf{l},\mathbf{l}'}\delta_{\mathbf{d},\mathbf{d}'}[\left|\overline{F_N(\mathbf{Q})}\right|^2-\overline{|F_N(\mathbf{Q})|^2}-\frac{1}{12}\overline{|B_{ld}|^2}]) \tag{6.26}$$

Comparing Eq. 6.25 and Eq. 6.26 we find that: 1) The nuclear coherent polarization is the same as the initial polarization. 2) The same is true for the incoherent scattering due to the random isotope distribution. 3) The nuclear incoherent scattering due to the random nuclear spin orientations has polarization $\mathbf{P}' = -1/3\mathbf{P}$ (for a random nuclear spin the associated Pauli matrix will 2/3 of the time point in the direction of $\hat{\sigma}_x$ and $\hat{\sigma}_y$ which according to Eq. 6.17 flips the spin).

For Vanadium, where there is only one isotope and coherent scattering is negligible, we find $\mathbf{P}' = -1/3\mathbf{P}$. There is however one catch. If the probability for multiple scattering is large one has to take into account that after two scattering one has: $\mathbf{P}'(2) = 1/9\mathbf{P}$, and so forth. The average polarization after a thick vanadium target is therefore a sum of different contributions.

6.2.2. Example: Polarizing Monochromator and Guides

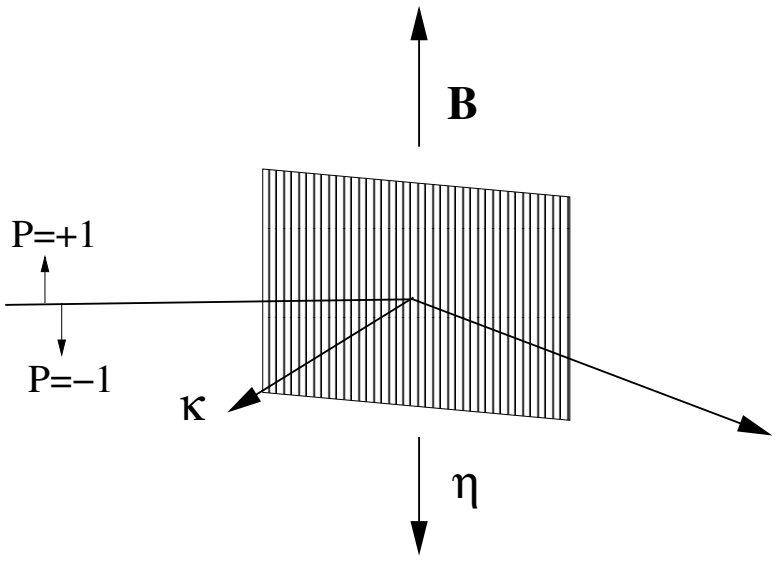


Figure 6.1.: Principle and geometry of a polarizing monochromator.

In a polarized monochromator and polarizing guides we have a ferromagnetic crystal in an external magnetic field. The scattering potential is now both nuclear (no internal direction) and magnetic (internal direction), so in general the outgoing polarization can be quite complex. However, as illustrated in Figure 6.1, the typical setup has many geometrical constraints: $\hat{\mathbf{N}} \cdot \hat{\mathbf{Q}} = 0$, $\hat{\mathbf{N}} \cdot \mathbf{P}_\perp = \hat{\mathbf{N}} \cdot \mathbf{P}$, and $\mathbf{Q} \times (\hat{\mathbf{N}} \times \mathbf{Q}) = \hat{\mathbf{N}}$, which simplifies the problem.

In [Lov84] the calculation for a centrosymmetric ferromagnetic crystal is done, and inserting the constraints above one finds ([Lov84], Eq. 10.96 and Eq. 10.110):

$$d\sigma/d\Omega = F_N^2 + 2F_N F_M(\mathbf{P} \cdot \hat{\mathbf{N}}) + F_M^2 \tag{6.27}$$

$$\mathbf{P}' d\sigma/d\Omega = \mathbf{P}[F_N^2 - F_M^2] + \hat{\mathbf{N}}[2F_N F_M + 2(\hat{\mathbf{N}} \cdot \mathbf{P})F_M^2] \tag{6.28}$$

> NB! Note that in [Wil88] Eq. 2.2.25 there is a minus in front of the second term in Eq. 6.27. We have not been able to understand this discrepancy, which is probably due to notation. Most other authors agree with the minus in front of the second term (e.g. Squires and Francis Tasset).

For a beam which is initially unpolarized we find the outgoing polarization to be:

$$\mathbf{P}' = \frac{\hat{\mathbf{N}} 2F_N F_M}{d\sigma/d\Omega} = \frac{2F_N F_M}{F_N^2 + F_M^2}\hat{\mathbf{N}}, \tag{6.29}$$

so that the beam is fully polarized along $\hat{\mathbf{N}}$ if $F_N = \pm F_M$.

What we use to characterize the polarizing monochromator in practice is not F_N and F_M, but instead the reflection probabilities $R_\uparrow$ and $R_\downarrow$ (for the reflection of interest).

If we assume that the reflection probabilities are directly proportional to the cross sections (with proportionality constant k), i.e., $R_\uparrow = kd\sigma/d\Omega(\mathbf{P} = +\hat{\mathbf{N}})$ and $R_\downarrow = kd\sigma/d\Omega(\mathbf{P} = -\hat{\mathbf{N}})$ then we can use Eq. 6.27 to determine F_N and F_M:

$$R_\uparrow = k(F_N + F_M)^2, \tag{6.30}$$

$$R_\downarrow = k(F_N - F_M)^2. \tag{6.31}$$

The values of $\sqrt{k}F_N$ and $\sqrt{k}F_M$ are then between -1 and +1 and unit less like the reflection probabilities. In the following we ignore k and just talk about F_N and F_M.

In principle there are four solutions for F_N and F_M, so in the code we currently choose the values where $F_N + F_M = +\sqrt{R_\uparrow}$ and $F_N - F_M = +\sqrt{R_\downarrow}$ (so that $F_N > 0$ and $F_N > F_M$). We then find:

$$F_N = \frac{\sqrt{R_\uparrow} + \sqrt{R_\downarrow}}{2}, \tag{6.32}$$

$$F_M = \frac{\sqrt{R_\uparrow} - \sqrt{R_\downarrow}}{2}. \tag{6.33}$$

When F_N and F_M are determined from these equations, Eq. 6.27 and Eq. 6.28 can easily be used to handle any situation.

This solution is both used for monochromators and guides.

It is not clear that this solution is correct. If we make a simple example with $R_\uparrow = 1$ and $R_\downarrow = 0.25$ then we could in principle have four solutions, but let us just quote the two where F_N is positive since the last two are found by inserting a minus before all the solutions and this does not change the physics. The two solutions are $F_N = 0.75, F_M = 0.25$ and $F_M = 0.75, F_N = 0.25$. All solutions gives the same cross section, but if the incoming beam is polarized (and only then) the outgoing beam will have two different polarization values, since $\mathbf{P}[F_N^2 - F_M^2]$ and $\hat{\mathbf{N}}2(\hat{\mathbf{N}} \cdot \mathbf{P})F_M^2$ are different for the two solutions. It seems that one needs some additional information to choose between the two solutions.

> NB! The simplifying geometry shown in Figure 6.1 only applies for the sides of the guide wall and not the top and bottom (assuming that the magnetizing field is pointing up or down), so there another set of equations should really be used.

The same physics could also be used for a polarizing powder or single crystal sample if F_N and F_M can be calculated with some other program, but one would have to use the general form of Eq. 6.27 and Eq. 6.28 without the simplifying geometrical constraints for monochromators and guides.

6.3. Magnetic fields: the precession algorithm and field framework

All magnetic-field-related polarisation components in McStas share a common run-time library, `pol-lib` (`mccode/nlib/share/pol-lib.c`/`.h`), which provides the spin-precession integrator and a small set of built-in field functions. A separate library, `ref-lib`, provides the standard supermirror reflectivity function (`StdReflecFunc`, section 6.2.2) used by the polarising mirrors and guides.

6.3.1. The magnetic field stack

Rather than each field-producing component computing precession locally and independently, `pol-lib` maintains a small per-particle *stack* of active magnetic fields, `_particle->mcMagnet`. A component that defines a field region (e.g. **Pol_Bfield**) *pushes* a description of its field onto this stack when a particle enters the region, using

```
mcmagnet_push(_particle, field_type, &rotation, &position, stopbit,
params[8])
```

where `field_type` selects one of the built-in field functions (section 6.3.3), `rotation`/`position` record the field region's placement in the lab frame (so the field can later be evaluated in the region's own local coordinates regardless of where in the instrument it sits), and

`params` holds up to 8 doubles of field-specific parameters (amplitude, extent, ...). A companion **Pol_Bfield_stop** component calls `mcmagnet_pop()` to remove the field from the stack once the particle leaves the region.

Because this is a stack rather than a single active field, **field regions may be nested or overlap**: any component placed between a `Pol_Bfield`/`Pol_Bfield_stop` pair – including another, independent field region – experiences the vector sum of every field currently on the stack (evaluated by `mcmagnet_get_field()`, which walks the whole stack, transforms each field into the lab frame, and adds the contributions). A field region may also be declared *closed* (`concentric=0`), in which case it pushes and pops its own field internally and no other components may be placed inside it – this is the simpler, non-nestable mode used when a single self-contained field region is all that is needed.

Whenever a particle is propagated through a region with an active field (technically: whenever `PROP_DT` is called while `_particle->mcMagnet` is non-`NULL`), the propagation macro transparently invokes the precession integrator described next, so that ordinary component code (slits, monitors, samples placed inside a field region) does not need to know anything about polarisation – the spin precesses correctly regardless of what else is happening to the particle during that propagation step.

6.3.2. Numerical spin precession: `SimpleNumMagnetPrecession`

The general-purpose precession routine, `SimpleNumMagnetPrecession()`, integrates Eq. 6.14 numerically along the particle's straight-line trajectory (gravity is *not* accounted for inside the field itself) for a total propagation time dt, using an *adaptive-step* scheme:

1. The field $\mathbf{B}_{\text{start}}$ is evaluated at the particle's current position (by querying the whole field stack, or – for the self-contained field components – a single field function).

2. A trial sub-step of length `mc_pol_initial_timestep` (default 10^{-5} s, capped at the remaining dt) is proposed, and the field $\mathbf{B}_{\text{end}}$ is evaluated at the particle's position at the end of that trial step.

3. If the angle between $\mathbf{B}_{\text{start}}$ and $\mathbf{B}_{\text{end}}$ exceeds a threshold, `mc_pol_angular_accuracy` (default 1°), the trial step is halved and re-evaluated; this repeats until the field direction changes by less than the threshold over the sub-step (or the step size underflows to `FLT_EPSILON`). This is what makes the scheme *adaptive*: in a slowly-varying or uniform field, the integrator quickly settles on large sub-steps (up to the 10^{-5} s cap), while in a rapidly rotating field (e.g. the `rotating` or `majorana` functions of section 6.3.3) it automatically refines the step until the field is well approximated as locally uniform.

4. Once an acceptable sub-step is found, the particle position and time are advanced by that sub-step, and the spin is precessed *about the mean field over the sub-step*, $\bar{\mathbf{B}} = \frac{1}{2}(\mathbf{B}_{\text{start}} + \mathbf{B}_{\text{end}})$, by the angle

$$\phi = \left(|\bar{\mathbf{B}}|\,\delta t\,\omega_L\right) \bmod 2\pi, \qquad \omega_L = -2\pi \times 29.16 \text{ MHz/T},$$

matching Eq. 6.15 applied over the sub-step, using a generic axis-angle rotation of (s_x, s_y, s_z) about $\bar{\mathbf{B}}$ (rather than the special-case z-axis form of Eq. 6.15, since $\bar{\mathbf{B}}$ may point in any direction).

5. This is repeated, consuming the remaining dt sub-step by sub-step, until the whole requested propagation time has been precessed over.

Positions/velocities/spin are tracked internally in the lab frame (fields are pushed to the stack together with the rotation/position needed to transform into and out of each field region's local frame) and the final spin is transformed back into the calling component's local frame before being returned. **Pol_tabled_field** (section 6.4) re-implements this identical algorithm internally (rather than going through the shared field stack), so that it can query its own tabulated/ interpolated field directly.

Two component-specific accuracy parameters, exposed via `mc_pol_set_timestep()` and `mc_pol_set_angular_accuracy()`, allow the initial sub-step size and the angular-accuracy threshold to be tuned instrument-wide if the defaults are not adequate for a particular field geometry.

6.3.3. Available field functions

The field pushed onto the stack by **Pol_Bfield** is one of the following built-in functions, selected by the integer `field_type` parameter (`Pol_FieldBox` and `Pol_constBfield` always use the constant-field case, evaluated in closed form rather than via the stack):

1 – constant $\mathbf{B} = (B_x, B_y, B_z)$, uniform throughout the region.

2 – rotating B starts as $(0, B_y, 0)$ at the entrance and rotates smoothly to $(B_y, 0, 0)$ at the far end (over the region's `zdepth`), following $B_x = B\sin(\frac{\pi}{2}z/L)$, $B_y = B\cos(\frac{\pi}{2}z/L)$.

3 – majorana a large component along x decreases linearly from $+B_x$ to $-B_x$ across the region (passing through zero at the centre – the classic "Majorana flip" geometry for studying spin-flip loss at a field zero-crossing), plus a small, constant transverse component along y.

6 – gradient linear interpolation of the full field vector between two given endpoint vectors $\mathbf{B}_1$ (at $z = -L/2$) and $\mathbf{B}_2$ (at $z = +L/2$) – a generalisation of the majorana case to an arbitrary field-vector gradient.

-1 – tabled used internally by **Pol_tabled_field**: the field is not one of the analytic functions above but is instead interpolated (via `interpolation-lib`, choosing between a `kdtree` or `regular`-grid interpolator) from an externally supplied point cloud file of (x, y, z, B_x, B_y, B_z) rows – this is the route to use for a realistic, measured or finite-element-computed field map.

4 (MSF) and 5 (RF) reserved identifiers for a modulated/RF spin flipper field; not implemented in the current release (a radio-frequency field is more commonly modelled explicitly via a rotating-frame field function or, for an idealised flipper, by simply mirroring the spin vector, see **Pol_SF_ideal** below).

New field functions can be added by writing a function with the signature of e.g. `const_magnetic_fiel` in `pol-lib.c` and adding a case to `magnetic_field_dispatcher()`; per the file header, this requires modifying the shared library rather than being an end-user-facing extension point.

6.4. Polarisation components in McStas

The components below can be grouped by role:

- **Setting and analysing polarisation** (unphysical, for testing/idealised use).
- **Polarising monochromators, mirrors and guides** (physical, reflectivity-based).
- **Magnetic field regions** (precession, using the framework of section 6.3).
- **Polarisation-sensitive monitors**.
- **Samples** affecting polarisation.

As of McStas 3.x these are found in the `optics` and `monitors` component categories (not a dedicated "polarisation" category); a component is polarisation-aware precisely when its `TRACE` section reads and/or writes the particle's spin state (s_x, s_y, s_z).

6.4.1. Setting and analysing polarisation

- **Set_pol:** An unphysical, zero-size component (drawn like an `Arm`) used to impose a polarisation state, in one of four modes selected by `randomOn` and `normalize`: (i) hard-code the polarisation to (p_x, p_y, p_z); (ii) set it to a random vector *on* the unit sphere (fully polarised, random direction); (iii) set it to a random vector *within* the unit sphere (random direction and degree of polarisation); (iv) hard-code the *direction* of (p_x, p_y, p_z) but normalise it to unit polarisation.
- **PolAnalyser_ideal:** An unphysical, ideal polarisation analyser. Given an analysis direction $\mathbf{m} = (m_x, m_y, m_z)$ ($|\mathbf{m}| \leq 1$ for an imperfect analyser), it reduces the neutron weight by the projection probability $\frac{1}{2}\big(1 + \mathbf{S} \cdot \mathbf{m}\big)$ and collapses the outgoing spin to $\mathbf{m}$, or absorbs the ray if that probability is zero or negative.
- **Pol_SF_ideal:** An idealised spin flipper: within an absorbing box, the polarisation vector is mirrored through the plane through the origin with normal (n_x, n_y, n_z) ($\mathbf{S}' = \mathbf{S} - 2(\mathbf{S} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$); rays missing the box pass through untouched. This is the natural building block for a spin-echo instrument's π-flip coils (section 6.5.4) when an idealised, 100%-efficient flipper is sufficient.

6.4.2. Polarising monochromators, mirrors and guides

These all implement the physics of section 6.2.2 (Eqs. 6.27–6.33): given reflectivities for spin up and down ($R_\uparrow$, $R_\downarrow$, either as `{R0,Qc,alpha,m,W}` parametrised curves via `ref-lib`'s `StdReflecFunc`, or as tabulated reflectivity files), the shared helper functions `GetMonoPolFNFM`, `GetMonoPolRefProb`, `SetMonoPolRefOut` and `SetMonoPolTransOut` (all in `pol-lib`) compute F_N, F_M, the reflection probability, and the outgoing polarisation for both the reflected and transmitted branches. Since a 2024 revision (E.B. Knudsen, P. Willendrup, H. Lee), all of these components treat a polarising reflection/transmission event as a projective quantum measurement along the mirror's quantisation axis: the in-plane spin components are explicitly zeroed ($s_x = s_z = 0$) at the point of interaction, before the new s_y is set by `SetMonoPolRefOut`/ `SetMonoPolTransOut` – reflecting that only the component of polarisation along the mirror's up direction survives a measurement-like interaction of this kind.

- **Monochromator_pol:** A flat, infinitely thin mosaic monochromator/analyser crystal (Gaussian mosaic and d-spread, billiard-ball reflection), the direct polarising analogue of **Monochromator_flat**, parametrised by `Rup`/`Rdown` (or `Q`/`DM` for the lattice spacing).

- **Pol_mirror:** A flat, infinitely thin, non-absorbing polarising mirror in the y-z plane, reflecting and/or transmitting according to `p_reflect` (use -1 to sample physically from the mirror's own reflectivity, or a fixed value to force a chosen reflect/transmit statistics split while re-weighting correctly).

- **Pol_bender:** A curved, multi-slit polarising bender (based on **Guide_curved**), with independently specifiable reflectivities for each of the four walls (top/bottom/left/right) and gravity support.

- **Pol_guide_mirror** / **Pol_guide_vmirror:** Straight rectangular guides with one (`_mirror`) or two, V-shaped (`_vmirror`) polarising supermirrors sitting on the diagonal(s) inside an otherwise ordinary (optionally non-polarising) guide. Setting the up and down reflectivities of the diagonal mirror equal turns either component into an unpolarised *frame-overlap mirror* instead – a deliberate dual use of the same geometry.

6.4.3. Magnetic field components

See section 6.3 for the shared precession algorithm and field functions used by these.

- **Pol_Bfield** / **Pol_Bfield_stop:** The general-purpose field region, supporting all of the field functions of section 6.3.3 and a box, cylindrical, spherical, or window-shaped extent. Used together as a `concentric` pair (`Pol_Bfield` pushes the field, other components may be placed freely inside, `Pol_Bfield_stop` pops it), or standalone as a closed region (`concentric=0`) that nothing else may be placed inside.

- **Pol_FieldBox:** A simple, self-contained, non-nestable box with a single uniform field (B_x, B_y, B_z), evaluated with a direct, single-step closed-form precession (Eq. 6.15) rather than the general stack/integrator – the lightest-weight choice for a plain guide field or spin-flip coil.

- **Pol_tabled_field:** A field region (box/cylinder/sphere, or an arbitrary OFF/PLY-geometry volume) whose field is interpolated from an externally supplied point-cloud file, using the same adaptive-step precession algorithm re-implemented internally (section 6.3.2). The interpolation backend (`kdtree` or `regular`-grid) can be chosen explicitly or left to an automatic CPU/GPU-appropriate default.

- **Pol_constBfield:** A rectangular, closed (non-nestable) box with a constant field along its local y-axis, evaluated with the closed-form solution of Eq. 6.15 directly (no field stack). Convenient for guide fields and simple spin flippers: rather than specifying B directly, `fliplambda`/`flipangle` let the field be specified as "the field that flips a neutron of this wavelength by this angle over this component's length" – exactly the `GetConstantField()` helper of `pol-lib`.

6.4.4. Polarisation-sensitive monitors

- **Pol_monitor:** Measures the projection of the polarisation onto a user-defined direction $\mathbf{m} = (m_x, m_y, m_z)$, i.e. $\mathbf{m} \cdot \mathbf{S}$, integrated over all detected rays.

- **PolLambda_monitor:** As **Pol_monitor**, resolved as a function of wavelength λ.

- **MeanPolLambda_monitor:** As **PolLambda_monitor**, but reporting the *mean* polarisation projection per wavelength bin (rather than the summed/integrated signal) – the natural monitor for a spin-echo polarisation-vs.-parameter scan (section 6.5.4).

- **PSD_monitor_4PI_spin:** A spherical, 4π position-sensitive monitor that additionally records the mean polarisation projection per pixel, for mapping how polarisation varies with scattering angle.

6.4.5. Samples and polarisation

Few McStas sample components currently propagate polarisation explicitly. The generic **Incoherent** component (the modern, general-purpose replacement for the historical, now-obsolete **V_sample**) can be parametrised to reproduce the classic depolarising-incoherent-scatterer result of section 6.2's nuclear-scattering example ($\mathbf{P}' = -\frac{1}{3}\mathbf{P}$ per single incoherent scattering event, from the random projection of nuclear spin onto $\hat{\sigma}_x$ and $\hat{\sigma}_y$); as noted there, this simple factor is only exact for a single scattering event, and multiple scattering must be handled with some care (consecutive depolarisation by $(-\frac{1}{3})^n$ is only valid if each scattering order can be isolated). Magnetic (Bragg) scattering from an ordered magnetic structure, using the magnetic structure factor F_M of section 6.1, is

not yet implemented as a general sample component; **Single_crystal** and **PowderN** presently model only nuclear scattering and do not read or write the spin state.

6.5. Test and example instruments

Every instrument below is found in the `Tests_polarization` category of the example library (browsable via `mcgui`'s "New from Template" menu, or `mcdoc`).

6.5.1. Component-level regression tests

These verify a polarising component reduces correctly to its non-polarising counterpart, or exercise one component's specific physics in isolation: *Test_Pol_Bender*, *Test_Pol_FieldBox*, *Test_Pol_Guide_mirror*, *Test_Pol_Guide_Vmirror*, *Test_Pol_Mirror*, *Test_Pol_MSF*, *Test_Pol_SF_ideal*, *Test_Pol_Set*, *Test_Pol_Tabled*, *Test_pol_ideal*, *Test_Magnetic_Constant*, *Test_Magnetic_Majorana*, and *Test_Magnetic_Rotation* (illustrating, respectively, the constant, majorana, and rotating field functions of section 6.3.3).

6.5.2. Test_Pol_TripleAxis

A complete example of a polarised triple-axis spectrometer: polarising monochromator and analyser, a vanadium (depolarising incoherent) sample, and a spin flipper – illustrating how the individual components of section 6.4 combine into a real instrument.

6.5.3. He3_spin_filter

Illustrates the use of a ^{3}He spin filter cell for beam or analysis-side polarisation.

6.5.4. A spin-echo example: SE_example

SE_example (and its variant *SE_example2*, both written for the 2010 PNCMI school in Delft) is a mock-up transmission spin-echo instrument, and a compact illustration of most of the machinery described in this chapter working together:

- **Pol_mirror** polarises the incoming beam and, symmetrically, analyses the scattered beam.
- **Pol_Bfield** regions define the two (oppositely-signed) precession guide fields on either side of the sample position, exactly the nested/concentric field-stack usage of section 6.3.1 (other components – the sample, monitors – sit *inside* each field region, between its `Pol_Bfield`/`Pol_Bfield_stop` pair).
- **MeanPolLambda_monitor** records the mean polarisation projection as a function of wavelength – the classic spin-echo readout, whose oscillation as a small guide-field perturbation `dBz` is scanned traces out the echo signal.

A typical scan takes the form

```
mcrun SE_example.instr dBz=-0.0001,0.0001 -N41 -n1e5
```

sweeping a small field imbalance `dBz` between the two precession regions and recording the resulting polarisation echo via **MeanPolLambda_monitor**. The instrument's own parameters (`POL_ANGLE`, `Bguide`, `Bflip`, `dBz`, `Lam`, `dLam`) let the polariser/analyser angle, the guide- and flipper-field strengths, and the source wavelength band be adjusted independently, and an optional incoherent scatterer (`SAMPLE=1`) can be inserted to study the effect of depolarising sample scattering on the echo.

7. The Union components

The **Union components** are a set of McStas components, originally developed by Mads Bertelsen [Ber17; BJL17], that take a fundamentally different approach to sample and sample-environment simulation than ordinary McStas components. This chapter explains the concepts behind the Union components and how to use them from an instrument file; the full parameter reference for every individual Union component is auto-generated from the component headers and found in the Component Manual's Union chapter.

7.1. Motivation

An ordinary McStas instrument is a linear sequence of self-contained components: a neutron ray enters a component, something happens to it (or not), and it moves on to the next component in the sequence. This is a good match for the intended beam path through an instrument, but it means that

- a single sample component must contain *all* the physics relevant to that sample – absorption, every scattering process, and any multiple scattering – as one monolithic block of code, and
- multiple scattering *between* physically separate parts of a sample environment (a sample inside a can, inside a cryostat, held by a sample stick, ...) is essentially impossible to capture, since each part would need to be a separate component and McStas components do not natively talk to each other.

This is a real limitation for complex sample environments, where multiple scattering between e.g. a powder sample and its aluminium container can be a significant background contribution.

The Union components solve this by splitting the sample simulation task into several cooperating component classes – *processes*, *materials*, *geometries* and a *master* – and performing the actual ray tracing for all of them together, inside the master component, with full native multiple scattering between every defined volume. Because physics (processes) and geometry are separated, adding a new scattering process is a comparatively small task, and existing processes can be freely combined and reused across different shapes.

It is entirely possible, and common, to mix ordinary McStas components with Union components in the same instrument file – the Union machinery only takes over at its `Union_master` components (section 7.2).

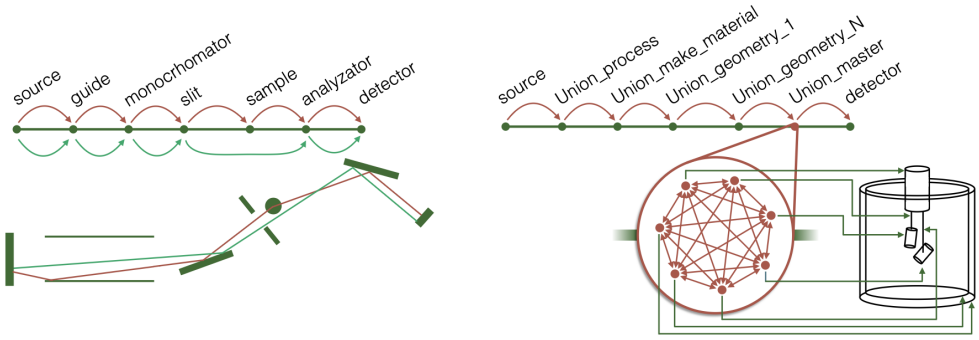


Figure 7.1.: Left: the linear succession of components in a traditional McStas instrument file. Right: the same instrument using Union components – all ray tracing for the sample environment happens inside a single master component, which works around the normal McStas component sequence to achieve native multiple scattering between an arbitrary number of volumes.

7.1.1. Volumes, priority and overlap

The central concept of the Union components is the *volume*: a geometrical shape (box, cylinder, sphere, cone or triangle mesh) combined with a material definition (absorption plus zero or more scattering processes), placed in space using the ordinary `AT`/`ROTATED` mechanism. Volumes may be placed in any order in the instrument file, and, unlike ordinary McStas geometry, volumes are allowed to *overlap*.

Each volume is given a unique `priority` value. Wherever two or more volumes overlap in space, the material of the volume with the *highest* priority applies. This single rule makes it straightforward to build geometries that would otherwise require careful, error-prone manual splitting into non-overlapping pieces: a hollow cryostat wall is simply an aluminium cylinder with a vacuum cylinder of higher priority placed inside it; a window is a low-priority vacuum region cut into an otherwise higher-priority wall; a layered sample can with several different powders is built from concentric or adjacent shapes, each its own volume and material.

7.2. Component classes

A Union simulation is assembled from components of several distinct classes, always used in the same overall order:

1. **Process components** – define a scattering process (section 7.2.2).

2. `Union_make_material` – collect processes and an absorption cross section into a named material (section 7.2.3).

3. **Geometry components** – place a volume with a given material in the simulation (section 7.2.4).

4. `Union_master` – performs the actual ray tracing for all volumes defined since the previous master (section 7.2.5).

In addition, optional **logger** (section 7.2.7) and **conditional** (section 7.2.8) components can be placed to record detailed information about what happens inside a master component, and **surface** components (section 7.2.6) can be attached to geometries to add reflection/refraction effects at their boundaries.

7.2.1. Loading the Union library

Union components need no setup or cleanup components. Each one loads the Union library itself, and the library keeps the internal bookkeeping lists (of processes, materials, geometries, loggers, ...) that the components communicate through, freeing them once the last Union component has finished.

An instrument that uses Union components must contain at least one `Union_master`; without one it fails to compile, with an error naming `Union_components_need_a_Union_master_in_the_instrume` Union components placed after the last `Union_master` have no effect, and the simulation warns about each of them when it finishes.

`Union_init` and `Union_stop`, which older instruments place first and last among their Union components, are no longer needed. They are kept without effect, apart from a deprecation warning, so that such instruments still compile. For the same reason every Union component still accepts the string parameter `init`, which older instruments set to the name of their `Union_init` component; its value is ignored.

7.2.2. Scattering processes

A process component has no physical shape of its own. It describes a single physical scattering mechanism: a function giving the inverse penetration depth (macroscopic cross section) as a function of incoming wavevector, and a function describing what happens to a ray in a scattering event. Table 7.1 lists the processes currently available; several are directly modelled on existing McStas sample components (e.g. `Incoherent_process` on `Incoherent`, `Powder_process` on `PowderN`, `Single_crystal_process` on `Single_crystal`), so users already familiar with those components will recognize most of the input parameters.

Every process has an `interact_fraction` setting parameter, in the range $[0, 1]$ or -1 to disable. When a material combines several processes, this can be used to artificially force a certain fraction of scattering events to that process, which is useful for improving statistics on a process that would otherwise rarely be sampled (an importance-sampling technique, entirely analogous to `p_interact` discussed below); the resulting ray weight is corrected automatically so the simulation remains unbiased. If left at -1 for every process in a material, the interact fractions default to the actual, physical scattering probabilities.

Component	Description
`Incoherent_process`	Isotropic, elastic incoherent scattering.
`Powder_process`	Bragg scattering from a powder (based on `PowderN`).
`Single_crystal_process`	Bragg scattering from a single crystal (based on `Single_crystal`); twinning is achieved by combining two or more instances with different orientations.
`Texture_process`	Single crystal scattering with an orientation distribution (texture) rather than a single fixed orientation.
`PhononSimple_process`	A single acoustic phonon branch.
`IncoherentPhonon_process`	Incoherent one-phonon scattering.
`AF_HB_1D_process`	Antiferromagnetic $S = 1/2$ 1D Heisenberg chain.
`Inhomogenous_incoherent_process`	Incoherent scattering with a spatially varying cross section.
`NCrystal_process`	Bragg and thermal-diffuse scattering via the external NCrystal library.
`Non_process`	A placeholder process with zero scattering probability.
`Template_process`	Documented template for writing a new process.

Table 7.1.: Scattering processes currently available for the Union components.

A minimal example defining an incoherent process for Vanadium:

```
COMPONENT Vanadium_incoherent = Incoherent_process(
    sigma=5.08, unit_cell_volume=13.827, packing_factor=1)
AT (0,0,0) ABSOLUTE
```

A process component's position in the instrument file is irrelevant to the physics (it has no shape); its *name* is what matters, since materials refer to processes by name.

Contributed processes: MCViNE kernels

A substantial set of further processes, ported from the MCViNE sample-scattering kernels, is contributed in the `contrib` component category rather than `union` (table 7.2); they follow the same Union process interface as the ones above and are used the same way. Each is paired with a standalone, non-Union component of the same name (without the `_process` suffix) sharing the same underlying kernel code.

A process does, however, use its rotation relative to ABSOLUTE to rotate its local frame. This can be used to create a small offset between two `Single_crystal_process` instances to describe a simple version of twinning. For that reason, always set the process rotation relative to ABSOLUTE.

Component	Description
`MCViNE_SQ_process`	Elastic isotropic scatterer with structure factor $S(\|Q\|)$ (expression or table).
`MCViNE_SQE_process`	Isotropic $S(Q, E)$ scatterer from an analytic expression or grid, with optional final-energy focusing.
`MCViNE_SvQ_process`	Elastic scatterer with a vector structure factor $S(Q_x, Q_y, Q_z)$ (expression or 3D grid).
`MCViNE_E_Q_process`	Isotropic dispersion: $S(Q, E) = S(Q)\delta(E - E(Q))$ with analytic $E(Q)$, $S(Q)$.
`MCViNE_E_vQ_process`	Single-crystal dispersion: $S(Q, E) = S(Q)\delta(E - E(Q))$ with analytic $E(Q_x, Q_y, Q_z)$.
`MCViNE_Broadened_E_Q_process`	Isotropic dispersion $E(Q)$ broadened by a Gaussian of Q-dependent width.
`MCViNE_LorentzianBroadened_E_Q_process`	Isotropic dispersion $E(Q)$ broadened by a Lorentzian of Q-dependent width.
`MCViNE_ConstantEnergyTransfer_process`	Isotropic scatterer with a fixed energy transfer.
`MCViNE_ConstantQE_process`	Scatterer with fixed momentum transfer $\|Q\|$ and energy transfer E.
`MCViNE_ConstantvQE_process`	Scatterer with a fixed momentum-transfer vector Q, weighted by a Gaussian in E.
`MCViNE_Phonon_CoherentInelastic_PolyXtal_process`	Coherent one-phonon scattering from a powder, using a full phonon dispersion (energies and polarizations) on a grid.
`MCViNE_Phonon_CoherentInelastic_SingleXtal_process`	Coherent one-phonon scattering from a single crystal, using a full phonon dispersion on a grid.
`MCViNE_Phonon_IncoherentElastic_process`	Incoherent elastic scattering with a Debye-Waller factor.
`MCViNE_Phonon_IncoherentInelastic_process`	One-phonon incoherent inelastic scattering from a phonon density of states, optional final-energy focusing.
`MCViNE_DGSSXRes_process`	Resolution kernel for direct-geometry single-crystal spectrometers (aims at a pixel and time-of-flight window).
`MCViNE_SANS2D_ongrid_process`	SANS from a 2D $S(Q_x, Q_y)$ map on a grid (beam along z).

Table 7.2.: Processes contributed from MCViNE, in the `contrib` component category.

7.2.3. Union_make_material

`Union_make_material` collects one or more previously defined processes into a named material, together with the absorption cross section (as the inverse penetration depth at

standard velocity, `my_absorption`, in m^{-1} at $v_0 = 2200$ m/s):

```
COMPONENT Al = Union_make_material(
    my_absorption=100*4*0.231/66.4,
    process_string="Al_incoherent,Al_powder")
AT (0,0,0) ABSOLUTE
```

If `process_string` is left unset, all processes defined since the previous `Union_make_material` are collected automatically; giving it explicitly is recommended, since it makes the resulting material self-documenting. Two material names are reserved and always available without being explicitly defined: `"Vacuum"` (no processes, no absorption) and `"Exit"` (behaves like vacuum, but see section 7.2.4 below). Setting `absorber=1` instead of giving a `process_string` creates a purely absorbing material with no scattering processes at all.

For the common case of a material that is purely NCrystal, with no other processes to combine it with, `Union_NCrystal_material` is a shortcut that creates the underlying NCrystal process and the material in a single component:

```
COMPONENT Cu = Union_NCrystal_material(cfg="Cu_sg225.ncmat")
AT (0,0,0) ABSOLUTE
```

Its absorption is derived automatically from the same NCrystal configuration string, so, unlike `Union_make_material`, it takes no `my_absorption` parameter. Use `NCrystal_process` together with `Union_make_material` instead whenever NCrystal scattering needs to be combined with other processes in the same material.

7.2.4. Geometry components

A geometry component places a volume of a given shape and material in the simulation, using the material name from `Union_make_material` and the ordinary `AT`/`ROTATED` keywords for position and orientation. The currently available shapes are `Union_box`, `Union_cylinder`, `Union_sphere`, `Union_cone` and `Union_mesh` (an arbitrary triangle mesh, for shapes not covered by the other primitives). No ray tracing happens in a geometry component itself – it only registers the volume with the following `Union_master`.

Every geometry component shares a common set of setting parameters, summarized in table 7.3, in addition to its own shape-specific dimensions (e.g. `radius`/`yheight` for `Union_cylinder`, `xwidth`/`yheight`/`zdepth` for `Union_box`).

```
COMPONENT cryostat_wall = Union_cylinder(
    radius=0.1, yheight=0.2, priority=10, material_string="Al")
AT (0,0,0) RELATIVE sample_position

COMPONENT cryostat_vacuum = Union_cylinder(
    radius=0.09, yheight=0.2, priority=11, material_string="Vacuum")
AT (0,0.01,0) RELATIVE sample_position
```

Here the vacuum cylinder (higher priority) hollows out the aluminium wall, and by displacing it 1 cm upward a solid bottom is left while the top stays open – a simple, direct application of the overlap-by-priority rule from section 7.1.1.

Parameter	Description
material_string	Name of a material from Union_make_material, or "Vacuum"/"Exit".
priority	Unique priority value; the volume with the highest priority wins where volumes overlap (section 7.1.1).
p_interact	Forces this probability [0-1] for a scattering event when a ray is inside the volume, regardless of path length (importance sampling; see note below).
visualize	Set to 0 to hide this volume in mcdisplay.
mask_string, mask_setting	Turn this volume into a *mask* restricting one or more other volumes (section 7.2.4).
number_of_activations	Number of subsequent Union_master components this volume should be simulated by (section 7.2.4).
surface_string parameters	Attach Union surface components (section 7.2.6) to this volume's faces. The exact parameter names are shape-specific: Union_sphere has a single surface (plus cut_surface for internal cuts from being overlapped); Union_box and Union_cylinder instead expose one parameter per face (e.g. plus_x_surface/minus_x_surface/...for the box, curved_surface/top_surface/bottom_surface for the cylinder), plus a convenience all_face_surface to set every face at once. See the Component Manual for the exact list per shape.
target_index/ target_x,y,z, focus_aw/focus_ah, focus_xw/focus_xh, focus_r	Standard McStas focusing parameters, used by any process assigned to this volume's material that supports focusing.

Table 7.3.: Setting parameters common to all Union geometry components.

p_interact and multiple scattering

Unlike p_interact in an ordinary McStas sample component, p_interact on a Union volume applies at *every* step of the multiple scattering loop, not just once. Setting it to 50%, for instance, gives a 25% chance of *two* scattering events in that volume. It

is therefore best kept well below 1; values close to 1 lead to very high probabilities of high-order multiple scattering and correspondingly large statistical weight corrections.

Masks

A mask volume restricts one or more other (*masked*) volumes: a ray only sees the masked volume's material where the masked volume and the mask volume(s) both cover the same space. This gives additional geometric freedom beyond what overlap-by-priority alone provides, and often reduces the number of volumes needed for an intricate shape. A masked volume can have several masks; `mask_setting` (`"All"` or `"Any"`, default `"All"`) controls whether every mask or just any one of them must cover a point for the masked volume to apply there. A mask volume is set via `mask_string` (a comma-separated list of the geometry names it masks) rather than `material_string`.

Exit volumes and number_of_activations

Normally, once a ray enters the region of space covered by a `Union_master`, it stays inside the Union ray-tracing loop until it escapes all defined volumes. Sometimes it is useful to break out of this loop early – for example to place an ordinary McStas monitor or component somewhere inside an otherwise Union-simulated geometry (a detector inside a detector tank, say). Assigning a volume the special material `"Exit"` achieves this: once a ray enters an exit volume, the current master stops and the *next* McStas component in the instrument sequence runs as normal; the ray does not automatically return to the Union simulation afterwards.

`number_of_activations` (default 1) controls how many subsequent `Union_master` components will simulate a given volume, which is useful when a static piece of geometry (a magnet, say) should be present across more than one master section of the instrument without redefining it.

7.2.5. Union_master

`Union_master` is where the actual ray tracing happens – it is the component that is placed into the ordinary linear McStas component sequence, and by default it simulates every volume defined since the previous master (or, for the first master, since the start of the instrument). If a volume is defined after the last `Union_master` in the file, it will never be simulated – remember to place a master after your geometry.

```
1 COMPONENT sample = Union_master()
2 AT (0,0,0) RELATIVE sample_position
```

Several of `Union_master`'s own setting parameters control the tagging system (section 7.2.9) and numerical safety limits; see its entry in the Component Manual's Union chapter for the complete list. A GPU-enabled variant, `Union_master_GPU`, is available for use with the OpenACC build (see the User Manual's chapter on running McStas, section on GPU acceleration); the regular `Union_master` does not run on GPU.

The `SCATTER` keyword is used internally, in a non-standard way, to mark every point where a ray moves from one volume to another, so that `mcdisplay` can show the resulting path. A variable `number_of_scattering_events` and an array `scattered_flag` (indexed by volume) are made available for use in an `EXTEND` block on the master, if you want to set your own custom flags based on where and how many times a ray scattered:

```
COMPONENT sample = Union_master()
AT (0,0,0) RELATIVE sample_position
EXTEND
%{
  if (number_of_scattering_events > 0) scattered_flag = 1; else
      scattered_flag = 0;
%}
```

7.2.6. Surfaces

Surface components describe physics that happens *at the boundary* of a volume rather than throughout its bulk – reflection and refraction, as opposed to the scattering and absorption described by processes. `Mirror_surface` implements standard supermirror-style specular reflectivity (the same $R_0/Q_c/\alpha/m/W$ parametrization, or a reflectivity data file, as used elsewhere in McStas); a `Template_surface` is provided for writing new surface types. A surface component is attached to a geometry's `surface` parameter (its outer faces) or `cut_surface` parameter (faces created where the volume is cut by being overlapped by a higher-priority volume), by name, exactly like a process is attached to a material. Refraction and reflection handling are toggled globally on `Union_master` via its `enable_refraction`/`enable_reflection` parameters (both on by default).

7.2.7. Logger components

Because so much of the simulation happens inside one master component, it is not possible to insert an ordinary McStas monitor *inside* the Union ray tracing to see what is going on there. Logger components fill this gap: attached to one or more volumes (and optionally further restricted to specific process names via `target_process`), a logger records information about every scattering event that happens there and writes it out as an ordinary McStas monitor data file. Leaving `target_geometry` empty logs *all* volumes, including ones not yet defined at the point the logger appears in the instrument file. Table 7.4 lists the currently available loggers; several further restrict what gets logged via an `order_total`/`order_volume`/`order_volume_process` parameter, so that only e.g. the first scattering event in a given volume is recorded.

A parallel set of *absorption loggers* (`Union_abs_logger_1D_space`, `_2D_space`, `_1D_time`, `_event`, `_nD`, and time-of-flight/ wavelength variants) record where and when absorption events happen instead of scattering events, which is often exactly what is needed for background studies of shielding and sample-environment components.

Component	Logs
`Union_logger_1D`	Time or $\|\mathbf{q}\|$ in a 1D histogram.
`Union_logger_2DQ`	(q_i, q_j) for a chosen pair of x, y, z.
`Union_logger_2D_kf`	As above, but the components of the final wavevector.
`Union_logger_2D_kf_time`	As above, with an additional time axis.
`Union_logger_2D_space`	Scattering position, 2D projection.
`Union_logger_2D_space_time`	As above, with an additional time axis.
`Union_logger_3D_space`	Scattering position in full 3D.

Table 7.4.: Logger components currently available for the Union components.

7.2.8. Conditional components

A logger on its own shows *where* scattering happens; a conditional component answers the complementary question of what *ends up* contributing to a particular part of the final result. A conditional modifies one or more named loggers (via `target_loggers`) so that they only actually record an event if the ray's final state – once it finally leaves the Union simulation – satisfies some condition. `Union_conditional_standard` filters on final time and/or energy range and/or total scattering order; `Union_conditional_PSD` additionally requires the ray to intersect a virtual position-sensitive detector. Several conditionals can be applied to the same logger, in which case all of them must be satisfied. Setting `master_tagging=1` instead applies the conditional to the tagging system (section 7.2.9) of the *next* master component, rather than to a specific logger.

Combining a position or scattering-vector logger with a conditional that filters, say, on a background region of the final detector signal, lets you directly inspect which volumes and processes actually contributed to that background – without having to re-run the simulation for each new question, as manual `EXTEND`-based tagging would require.

7.2.9. Tagging: ray histories

Independently of the logger system, `Union_master` can record a simple *history* for every ray: the ordered sequence of volumes entered and scattering processes undergone. All rays sharing the same sequence of volumes/processes count as the same history; histories are collected and, at the end of the simulation, written to a file (by default `Union_history.dat`) sorted by the total intensity they represent. An excerpt for a simple aluminium-can-with-powder setup:

```
1 V0 : Surrounding vacuum
2 V1 : powder_container          Material: Al        P0: Al_incoherent P1:
    Al_Powder
3 V2 : powder_inside_container   Material: Cu_powder  P0: Cu_incoherent  P1:
    Cu_powder
4 ------ Histories sorted after intensity ------
5 12221626  I=4.188166E-06   V0
```

```
 1882051   I=1.052731E-06    V0 -> V1 -> V2 -> V1 -> V0
 1517013   I=6.213315E-07    V0 -> V1 -> V0
  188661   I=8.043799E-08    V0 -> V1 -> V2 -> P0 -> V1 -> V0
```

`VX` refers to volume number X, `PX` to process number X *within whatever volume precedes it* in the string (so `P0` after `V1` and `P0` after `V2` are generally different physical processes, if V1 and V2 have different materials). This is invaluable for background investigations: to estimate how much intensity originates from scattering in a particular volume, search the history file for occurrences of `"VX -> P"`; to find only the part of that which reaches a particular detector, combine it with an exit volume placed there and search for histories ending in that volume's number. Tagging is controlled by `Union_master`'s `enable_tagging` and `history_limit` parameters (the latter caps memory use by limiting the number of distinct histories retained).

7.3. The propagation algorithm

This section gives a conceptual outline of how `Union_master` actually propagates rays; see the original thesis [Ber17] for the full derivation and pseudocode.

7.3.1. Basic Monte Carlo step

Within a volume, the distance to the next scattering position is drawn from the usual exponential distribution set by the total inverse penetration depth (sum over the volume's absorption and all its processes' scattering contributions). If this distance is less than the distance to the volume's boundary, a scattering event occurs there; the specific process is chosen by a weighted Monte Carlo choice between the processes active in the current material (or forced via `interact_fraction`, in which case the ray weight is corrected to remain unbiased). Otherwise, the ray is propagated to the boundary and the simulation continues in whichever volume lies on the other side.

7.3.2. Intersection and destination lists

Naively, determining "the other side" of a boundary would require testing every other defined volume for intersection and containment, for every propagation step – prohibitively expensive once more than a handful of volumes are involved. Instead, in the `initialize` section of each `Union_master`, the whole ensemble of volumes is analysed once to build, for every volume n:

- an *intersection list* I_n: the (usually small) set of other volumes that actually need an intersection test while the ray is inside volume n – volumes with lower priority, or volumes properly nested inside another candidate, are pruned out;
- a *destinations list* D_n: the set of volumes the ray could possibly end up in immediately after leaving volume n through its own boundary, used to resolve exactly which volume that is when several overlap there.

This turns an $O(N)$-per-step cost (checking every volume) into something close to $O(1)$ in typical geometries, and is what makes simulating sample environments built from dozens of volumes computationally practical. Figure 7.2 sketches the resulting behaviour for a simple case.

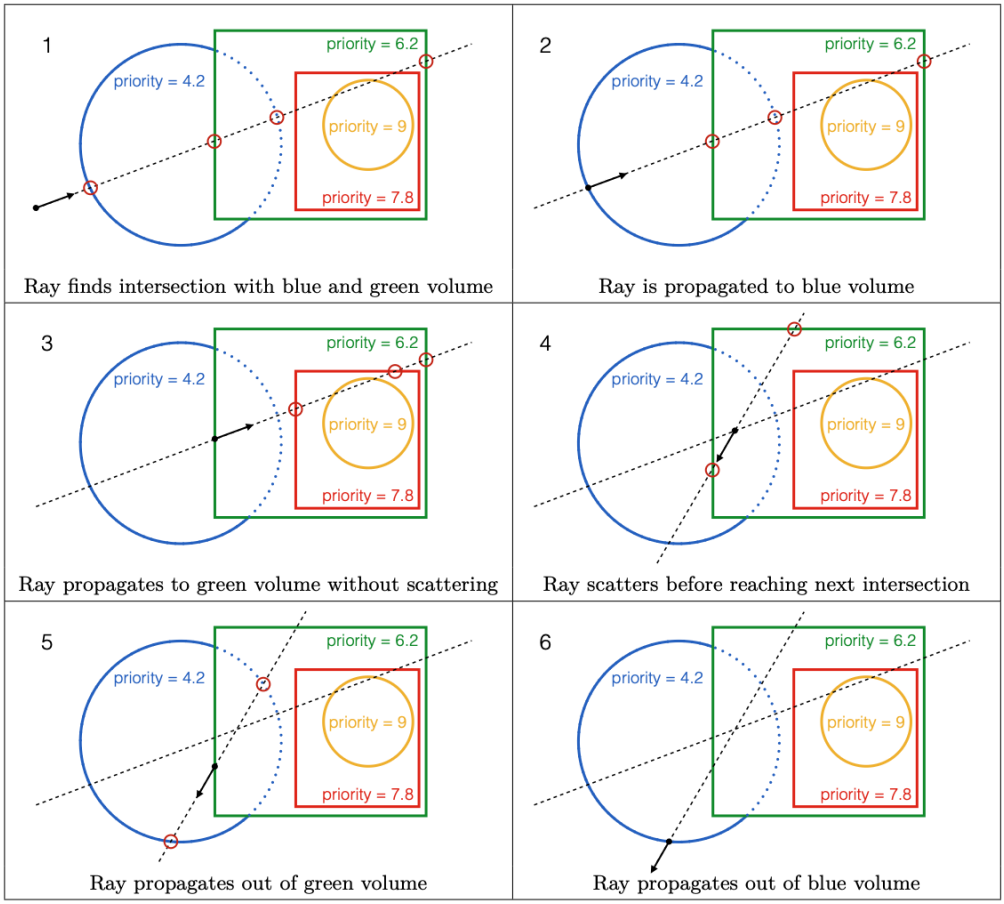


Figure 7.2.: Illustration of the Union propagation algorithm for a single scattering event between two overlapping volumes. Only intersections actually relevant given the ray's current volume are calculated at each step (the orange volume, never reachable from the ray's actual path here, is never tested).

7.3.3. Known limitations

- **Coincident surfaces.** If two volume boundaries coincide exactly, the algorithm cannot reliably determine which volume a ray enters (the decision is essentially made by floating-point rounding). It is safe, and normal, to *overlap* volumes – but two volumes should never be placed so that a face merely touches without any overlap or gap; leaving even a micron-scale gap or overlap avoids the problem

entirely.

- **No gravity.** The underlying intersection algorithms are linear, so Union volumes do not support gravity; propagation inside a `Union_master` is always a straight line even if `--gravity` is enabled for the rest of the instrument.

- **GPU.** The default `Union_master` is CPU-only (`NOACC`); use `Union_master_GPU` for OpenACC builds (see the User Manual's chapter on running McStas, section on GPU acceleration).

7.4. A complete example

The following excerpt shows the pieces from this chapter assembled into a working instrument fragment: an aluminium sample can holding an iron sample, both with incoherent and powder scattering.

```
COMPONENT Al_inc = Incoherent_process(sigma=4*0.0082, unit_cell_volume
    =66.4)
AT (0,0,0) ABSOLUTE

COMPONENT Al_powder = Powder_process(reflections="Al.laz")
AT (0,0,0) ABSOLUTE

COMPONENT Al = Union_make_material(my_absorption=100*4*0.231/66.4,
    process_string="Al_inc,Al_powder")
AT (0,0,0) ABSOLUTE

COMPONENT Fe_inc = Incoherent_process(sigma=2*0.4, unit_cell_volume=24.04)
AT (0,0,0) ABSOLUTE

COMPONENT Fe_powder = Powder_process(reflections="Fe.laz")
AT (0,0,0) ABSOLUTE

COMPONENT Fe = Union_make_material(my_absorption=100*2*2.56/24.04,
    process_string="Fe_inc,Fe_powder")
AT (0,0,0) ABSOLUTE

/* ... source, guide etc. as normal ... */

COMPONENT sample_position = Arm()
AT (0,0,12.5) RELATIVE Guide

COMPONENT can = Union_cylinder(radius=0.006, yheight=0.04,
    priority=1, material_string="Al")
AT (0,0,0) RELATIVE sample_position

COMPONENT can_vacuum = Union_cylinder(radius=0.005, yheight=0.038,
    priority=2, material_string="Vacuum")
AT (0,0,0) RELATIVE sample_position

COMPONENT sample = Union_cylinder(radius=0.004, yheight=0.03,
```

```
35      priority=3, material_string="Fe")
36 AT (0,0,0) RELATIVE sample_position
37
38 COMPONENT master = Union_master()
39 AT (0,0,0) RELATIVE sample_position
40
41 COMPONENT Banana_monitor = Monitor_nD(radius=1, yheight=0.1,
42      options="banana, theta limits=[20,170], bins=200", filename="banana.dat
           ")
43 AT (0,0,0) RELATIVE sample_position
```

7.5. Further examples and worked tutorials

The example instrument library ships several dedicated Union categories: `Union_demos` (including `Manual_example`, essentially the worked example above), `Union_sample_environments`, `Union_validation` (comparisons against the traditional McStas components each process is based on) and `Tests_union` (unit-style tests of individual features). See the User Manual's chapter on instrument examples for how to browse these. A number of the individual component headers, documented in the Component Manual's Union chapter, contain further short usage examples.

For a more gradual, tutorial-style introduction covering geometry, masks, loggers, conditionals, exit volumes and tagging one step at a time, the McStasScript documentation's Union tutorial series is a useful complement to this chapter [Mcsc]. It is written using McStasScript's Python interface rather than the plain McStas instrument-file syntax used throughout this manual, but the underlying Union concepts – and the components themselves – are identical, so the explanations translate directly.

8. Links to other computing codes

This chapter provides information on interoperability of McStas with other codes, and provide links to other sources of information on this topic.

8.1. McStas and MANTID

As of McStas 2.1 and MANTID 3.2, the teams of the two codes provide a mechanism to transfer simulated McStas events and monitor data to MANTID, via NeXus files.

8.1.1. System requirements

To enable the software link, you need the following codes installed on your system:

- McStas 2.1 or newer
- The HDF5 library from `https://www.hdfgroup.org` (also in conda-forge, Homebrew and the Linux distributions)
- Mantid 3.2 or newer
- On Mac OS X 10.8 and newer you may need to install one of the below compilers, as the default clang on OS X causes problems for the link, especially when used together with MPI
  - Intel C
  - gcc from `http://hpc.sourceforge.net`

8.1.2. Requirements for the instrument file

A special naming convention in the instrument file is needed for the automatic transfer of geometry to a Mantid IDF file:

- The location of the source must be indicated by a component named sourceMantid. Note that in the case of a curved instrument geometry, you should probably add an Arm where Mantid 'expects' the source to be, i.e.: defining a location displaced the correct source-sample distance, parallel to the incoming beam direction at the sample.
- The location of the sample must be indicated by a component named sampleMantid

- One or more Monitor_nD components need to be added in either rectangular- or cylindrical geometry and with a set of special flags, as shown below
- Rectangular monitor

```
COMPONENT nD_Mantid_0 = Monitor_nD(
 options ="mantid square x limits=[-0.2 0.2] bins=128 y limits
     =[-0.2 0.2] bins=128, neutron pixel t, list all neutrons",
 xmin = -0.2,
 xmax = 0.2,
 ymin = -0.2,
 ymax = 0.2,
 restore_neutron = 1,
 filename = "bank01_events.dat")
AT (0, 0, 3.2) RELATIVE sampleMantid
```

- Cylindrical monitor

```
COMPONENT nD_Mantid_01 = Monitor_nD(xwidth=(4.0-0.0005-0.00002)*2,
      yheight=3,
 options="mantid banana, theta limits=[-73.36735 73.36765] bins
     =100, y limits=[-1.5 1.5] bins=300, neutron pixel t, list all
     neutrons", restore_neutron=1)
AT (0,0,0) RELATIVE center_det
```

The two instruments `templateSANS_Mantid.instr` and `ILL_H16_IN5_Mantid.instr` have these features enabled.

Other, ordinary McStas monitors will also be visible in the resulting Mantid workspace, but will not be easily processible using the Mantid TOF data reduction schemes.

8.1.3. Compiling and running your simulation for Mantid output

Geometry information in Mantid is handled via a so-called Instrument Definition File (IDF) in xml-format, possibly embedded in a NeXus file. The creation of the IDF and related NeXus file is handled in the following steps:

1. First of all, enable NeXus in the compilation process:

```
export MCSTAS_CFLAGS="-g -lm -O2 -DUSE_NEXUS -lhdf5"
```

2. Compile the instrument via the mcrun utility (add `--mpi` if you need parallelization support):

```
mcrun -c ILL_H16_IN5_Mantid.instr -n0
```

3. Generate the IDF via the mcdisplay utility (and press enter until the simulation runs):

```
mcdisplay --format=Mantid ILL_H16_IN5_Mantid.instr -n0
```

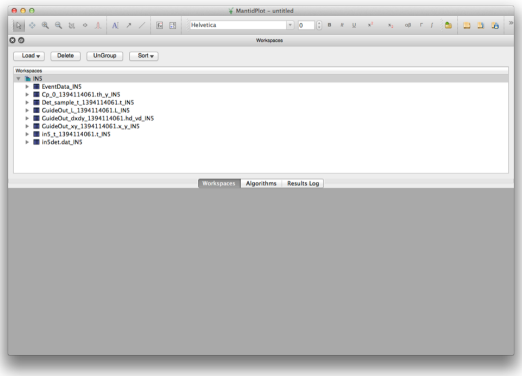
4. Finally, run a simulation with NeXus output:

```
mcrun --format=NeXus ILL_H16_IN5_Mantid.instr
```

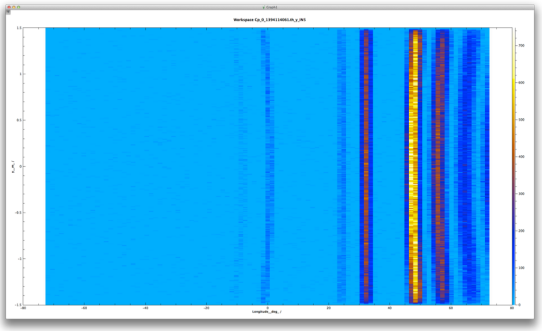
8.1.4. Looking at instrument output in Mantid

In Mantid, your new NeXus file should behave more or less as usual, that is

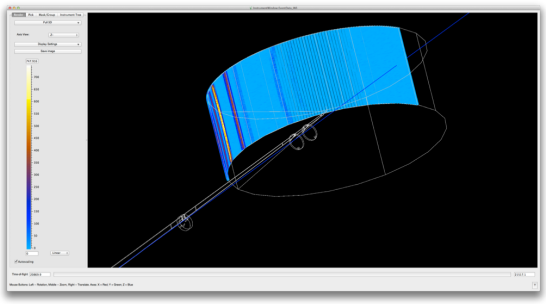
- Start by loading the file using either the Load button or the algorithm LoadMcStas to load the event file.
- After a succesful load, you should have a workspace group with this content



- Plot the histogram event data stored in the monitor Cp_0...



- Plot the TOF events on the full instrument geometry. In this step a bit of zooming and translation may be needed to show a nice view of the instrument.



The instrument `ILL_H16_IN5_Mantid.instr` included in McStas 2.1 will load and present simulated event-data, but the current time-definition of the event data does not comply fully with Mantid for data-reduction purposes - this should be corrected for the next McStas release.

8.2. McStas and MCNP(X)

As of McStas 2.1 we provide a a series of mechanisms to interchange simulated events between McStas and MCNP or MCNPX:

1. Typically, the intensity parameters in our McStas source components are derived from MCNP(X) simulations, though parametric fits to MCNP*tallies*.

2. The PTRAC method to read MCNP(X) events into McStas has been available since McStas 1.10 and is a contribution from Chama Hennane, ENSIMAG and Emmanuel Farhi, ILL.

3. The SSR/SSW method to send events via the MCNP "source surface" mechanism between both codes has been available since McStas 2.0. This work has mainly been done by Esben Klinkby, DTU Nutech, with contributions from Erik Knudsen and Peter Willendrup, both DTU Physics.

4. With access to an MCNP source-code license and a patch developed by Esben Klinkby, DTU Nutech, it is further possible to directly compile and link a McStas instrument with MCNP. The method is however quite speed-limited at present, but we envision releasing an improved version later. If interested, please contact Esben Klinkby directly.

Especially method 3 gives a powerful combination with the *scatter logging* mechanism discussed in the paper [KKW14], allowing to estimate γ dose-rates in MCNP from neutron intensity lost in neutron optics, i.e. the non-transmitted fraction of the McStas neutron beam (and corresponding intensity/weight). Through the McStas example instruments and also the McStas share at `http://mcstas.org/download/share` we provide a couple of examples of the use of the *scatter logger*

9. The component library: Abstract

This chapter presents an abstract of the McStas component library's structure and categories. It intentionally does *not* attempt to enumerate every component with its parameter list – the library is large and changes between releases, so any hand-maintained table here would inevitably drift out of date (as earlier editions of this manual did). Instead, each component category chapter that follows (Sources, Optics, Samples, Monitors, Misc, ...) documents a curated selection of representative components in depth, with their full, current parameter lists pulled directly from the component source's McDoc header at build time (see section 5.7 for the McDoc comment format). For the complete, authoritative, always-current list of every component and its parameters, use the `mcdoc` front-end:

```
mcdoc
mcdoc searchterm
mcdoc file.comp
```

The first form generates and opens a browsable *index.html* catalog of the whole library (both the standard installation and any local/custom components); the second searches for and displays documentation for all components matching *searchterm*; the third displays the documentation of one specific component file. See section 4.5.8 in the User Manual for the full `mcdoc` reference.

9.1. Component categories

The library (located under the `mcstas-comps` directory of the McStas source/installation tree, see section 4.3.2 in the User Manual for how `mcstas` locates it) is organised into the following top-level categories:

sources Components that define the initial neutron state – see the Sources chapter of the Component Manual.

optics Guides, slits, choppers, monochromators, mirrors, and other beam-manipulating components.

samples Components modelling matter placed in the beam for scattering studies (powders, single crystals, SANS models, ...) – see the Samples chapter of the Component Manual.

monitors Components that record/histogram the neutron beam without otherwise affecting it.

misc Components that do not fit the other categories, e.g. shape primitives, the `Progress_bar`, and `MCPL_input`/`MCPL_output` for reading/writing portable neutron event files.

union The Union framework (contributed by Mads Bertelsen): a set of process/geometry components that can be freely combined to describe complex, possibly nested and overlapping sample environments with multiple scattering, decoupling the description of *where* matter is from *what* it does to the neutron. See the worked example in the User Manual's chapter on instrument examples (Union_demos) and the samples chapter.

sasmodels Small-angle scattering form factors generated from the SasView/sasmodels project, exposed as individual `SasView_*` components (the older, generic `SasView_model` component is now obsolete).

contrib Components contributed by McStas users. The McStas core team provides the same technical support as for other categories, but authored and maintained the code less directly; contributed components are generally reliable, but see each component's own validation/bugs notes. The category also holds the MCViNE-derived sample kernels, which are available both as ordinary sample components and as Union processes. Components in `contrib` that mature and gain wide use are periodically promoted into one of the core categories above (e.g. several guide, chopper and polarisation components originally contributed have since moved into `optics`) – always check with `mcdoc` rather than assuming a component's category from an older document.

obsolete Components that were renamed or superseded. They are kept, and continue to work, purely for backwards compatibility with existing instrument files; new instrument development should avoid them (`mcdoc` will point to the current replacement where one exists).

parked Components that are kept in the source tree (e.g. pending further validation, a rewrite, or reinstatement) but excluded from the regular build/test cycle.

9.2. Data files

Some components require external data files, e.g. lattice crystallographic definitions for Laue and powder pattern diffraction, $S(q,\omega)$ tables for inelastic scattering, or transmission/reflectivity curves. Such files distributed with McStas are located in the `data` subdirectory of the library and are accessible to any component using the `read_table-lib` shared library (see section B.2) without needing local copies.

9.3. Component and instrument examples

An overview of the example instrument library, organised by facility and category, is given in the User Manual's chapter on instrument examples. Every `.instr` file there is

MCSTAS/data	Description
*.lau	Laue pattern file, as issued from Crystallographica. For use with Single_crystal, PowderN, and Isotropic_Sqw. Data: [h k l Mult. d-space 2Theta F-squared]
*.laz	Powder pattern file, as obtained from Lazy/ICSD. For use with PowderN, Isotropic_Sqw and possibly Single_crystal.
*.trm	transmission file, typically for monochromator crystals and filters. Data: [k (Angs-1) , Transmission (0-1)]
*.rfl	reflectivity file, typically for mirrors and monochromator crystals. Data: [k (Angs-1) , Reflectivity (0-1)]
*.sqw	$S(q,\omega)$ files for Isotropic_Sqw component. Data: [q] [ω] [$S(q,\omega)$]

Table 9.1.: Data files of the McStas library.

discoverable via `mcgui`'s **New from Template** menu, or via `mcdoc`.

10. Instrument examples

In this section, we present a few typical instruments from the McStas example library. We then give a longer description of three selected worked examples: a simple sample test instrument (10.2), the historical Risø triple-axis spectrometer TAS1 (10.3), and the ISIS time-of-flight spectrometer PRISMA (10.4).

These instrument files are included in the McStas distribution in the `examples/` directory (`mcstas-comps/examples` in the source tree), organised by facility/category. Any of them can be loaded from `mcgui`'s **New from Template** menu (known as "Neutron site" in older McStas releases, see section 4.2): selecting one copies the instrument into your working directory, ready to edit, run, or trace.

The list below is a curated tour, not an exhaustive catalogue – new example instruments are added continuously as facilities contribute instrument descriptions. Use `mcdoc` (section 4.5.8) or browse the `examples/` directory directly for the full, current list.

10.1. A quick tour of instrument examples

10.1.1. Templates

The `Templates` category is the recommended starting point for new instrument work: skeleton instruments to copy and adapt, covering the most common instrument classes – `template` and `template_simple` (bare instrument skeletons), `templateTAS` (triple-axis, including a basic in-plane UB-matrix transformation for resolution estimation), `templateTOF` and `templateDIFF` (time-of-flight and powder diffraction), `templateSANS`/`templateSANS` and `templateSasView` (small-angle scattering, the latter using the `SasView_model` interface to SasView/sasmodels form factors), `templateLaue` and `templateNMX`/`templateNMX_TOF` (single-crystal Laue diffraction), and `Demo_shape_primitives` (illustrates the general-purpose shape/geometry components). Instrument parameters are documented in each file's header (readable via `mcdoc`) and are designed to cover most instruments of the corresponding class.

10.1.2. Risoe

Historical Risø National Laboratory instruments, including the TAS1 triple-axis spectrometer family described in detail in section 10.3 below. TAS1 was decommissioned along with the DR3 reactor, but its detailed McStas model remains one of the most thoroughly validated example instruments in the distribution (see the model-vs-measurement comparisons in section 10.3.1), and it constitutes the default regression test series for the McStas distribution.

10.1.3. ISIS

Instruments modelled on ISIS spallation-source beamlines, several using the `ISIS_moderator` source component (built from MCNP moderator calculations): `ISIS_CRISP` and `ISIS_SANS2d` (reflectometry and SANS), `ISIS_HET`, `ISIS_LET`, `ISIS_MERLIN` and `ISIS_TOSCA_preupgrade` (chopper spectrometers), `ISIS_GEM` and `ISIS_OSIRIS` (powder diffraction and backscattering), and `ISIS_Prisma2`, a simple time-of-flight instrument with a RITA-style analyser described in detail in section 10.4 below.

10.1.4. ILL

Cold and thermal guide models for a large number of ILL instruments, several based on detailed technical drawings, e.g. `ILL_H15`, `ILL_H16`, `ILL_H22` and `ILL_H53` (guide systems) feeding instrument-specific models such as `ILL_H15_IN6` (a hybrid time-of-flight spectrometer with three monochromators and a Fermi chopper; a usage example of the `GROUP` and `SPLIT` keywords, see sections 5.4.2 and 5.4.6 in the Component Manual), `ILL_H15_D11` and `ILL_H512_D22` (SANS), `ILL_H22_D1A`/`ILL_H22_D1B` and `ILL_D2B` (powder diffraction), `ILL_H53_IN14` and `ILL_H142_IN12` (triple-axis), `ILL_IN5` and `ILL_IN4` (time-of-flight), and `ILL_H22_VIVALDI`/`ILL_H143_LADI` (Laue diffraction).

10.1.5. ESS

Design studies for the European Spallation Source, built around the `ESS_butterfly` moderator/source component, e.g. `ESS_IN5_reprate` (a cold chopper multi-frame spectrometer for the long-pulsed ESS timing structure) and `ESS_BEER_MCPL` (illustrating MCPL-based event exchange between simulation stages, see the User Manual section on MCPL/event files).

10.1.6. Union_demos, Union_sample_environments, Union_validation

Worked examples of the Union framework (contributed by Mads Bertelsen), which separates sample/environment *geometry* from *physical process* definitions and supports multiple scattering within arbitrarily complex, nested sample-environment assemblies. `Union_demos/Demonstration` is a good first read: several powder samples in cans, a sample holder, and a surrounding cryostat, illustrating the syntax and possibilities of the Union components. See the Component Manual chapter on samples for the underlying Union component reference.

10.1.7. NCrystal

Examples using the `NCrystal_sample` component, interfacing the external NCrystal library for physically detailed elastic and inelastic scattering from single crystals, powders, and (to some extent) liquids, based on material scattering kernels.

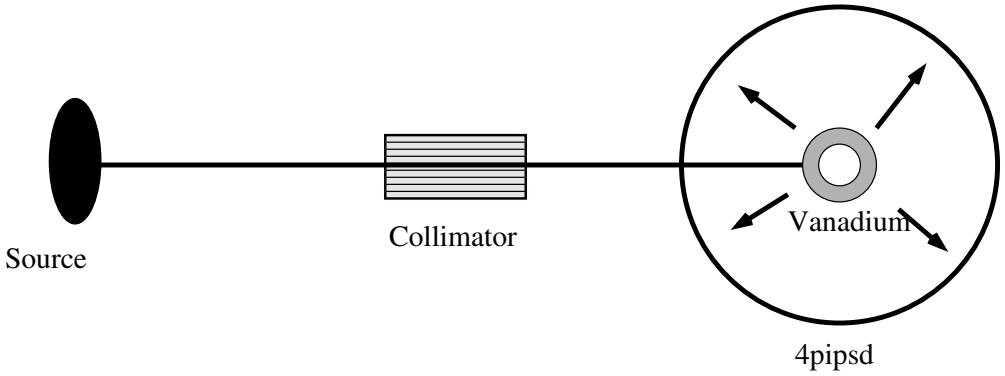


Figure 10.1.: A sketch of the test instrument for the component V_sample.

10.1.8. Tests_* categories

A large set of `Tests_sources`, `Tests_optics`, `Tests_samples`, `Tests_monitors`, `Tests_polarizatio` `Tests_union`, `Tests_grammar`, `Tests_RNG`, `Tests_MCPL_etc` and `Tests_other` instruments exercise individual components or language features in isolation. These are the backbone of the McStas regression test suite (run via `mctest`, see section 4.5.9), but are equally useful as minimal starting points when learning how a specific component behaves. `Tests_samples/Samples_vanadium` is described in detail in section 10.2 below.

10.1.9. Other facility categories

Further contributed instrument models exist for BNL, DTU, FZ_Jülich, HZB, HighNESS, LLB, Necsa, PSI, SINE2020, SNS, TRIGA and TU Delft, alongside a `Mantid` category (instruments paired with Mantid IDF export via `mcdisplay-mantid_xml`, see section 4.5.5) and a `Tools` category (utility instruments, e.g. for reading back third-party event files via MCPL). Use `mcdoc` to search across all categories by keyword, component used, or facility name.

10.2. A test instrument for the component V_sample

This is one of many test instruments written with the purpose of testing the individual components. We have picked this instrument both to present an example test instrument and because it despite its simplicity has produced quite non-trivial results.

The instrument, `Tests_samples/Samples_vanadium`, consists of a narrow source, a 60' collimator, a V-sample shaped as a hollow cylinder with height 15 mm, inner diameter 16 mm, and outer diameter 24 mm at a distance of 1 m from the source. The sample is in turn surrounded by an unphysical 4π-PSD monitor with 50×100 pixels and a radius of 10^6 m. The set-up is shown in figure 10.1.

10.2.1. Scattering from the V-sample test instrument

In figure 10.2, we present the radial distribution of the scattering from an evenly illuminated V-sample, as seen by a spherical PSD. It is interesting to note that the variation in the scattering intensity is as large as 10%. This is an effect of anisotropic attenuation of the beam in the cylindrical sample.

10.3. The triple axis spectrometer TAS1

With this instrument definition, we have tried to create a very detailed model of the conventional cold-source triple-axis spectrometer TAS1 at the now closed neutron source DR3 of Risø National Laboratory. Except for the cold source itself, all components used have quite realistic properties. Furthermore, the overall geometry of the instrument has been adapted from the detailed technical drawings of the real spectrometer. The TAS 1 simulation was the first detailed work performed with the McStas package. For further details see reference [ACL98]. The instrument family lives under `Risoe/` in the example library, e.g. `Risoe/TAS1_Vana` and `Risoe/TAS1_Powder`, and is also known by its historical name `linup-*` in older documentation.

At the spectrometer, the channel from the cold source to the monochromator is asymmetric, since the first part of the channel is shared with other instruments. In the instrument definition, this is represented by three slits. For the cold source, we use a flat energy distribution (component **Source_flat**) focusing on the third slit.

The real monochromator consist of seven blades, vertically focusing on the sample. The angle of curvature is constant so that the focusing is perfect at 5.0 meV (20.0 meV for 2nd order reflections) for a 1×1 cm^2 sample. This is modeled directly in the instrument definition using seven **Monochromator** components. The mosaicity of the pyrolytic graphite crystals is nominally 30' (FWHM) in both directions. However, the simulations indicated that the horisontal mosaicities of both monochromator and analyser were more likely 45'. This was used for all mosaicities in the final instrument definition.

The monochromator scattering angle, in effect determining the incoming neutron energy, is for the real spectrometer fixed by four holes in the shielding, corresponding to the energies 3.6, 5.0, 7.2, and 13.7 meV for first order neutrons. In the instrument definition, we have adapted the angle corresponding to 5.0 meV in order to test the simulations against measurements performed on the spectrometer.

The width of the exit channel from the monochromator may be narrowed down from initially 40 mm to 20 mm by an insert piece. In the simulations, we have chosen the 20 mm option and modeled the channel with two slits to match the experimental set-up.

In the test experiments, we used two standard samples: An Al_2O_3 powder sample and a vanadium sample. The instrument definitions use either of these samples of the correct size. Both samples are chosen to focus on the opening aperture of collimator 2 (the one between the sample and the analyser). Two slits, one before and one after the sample, are in the instrument definition set to the opening values which were used in the experiments.

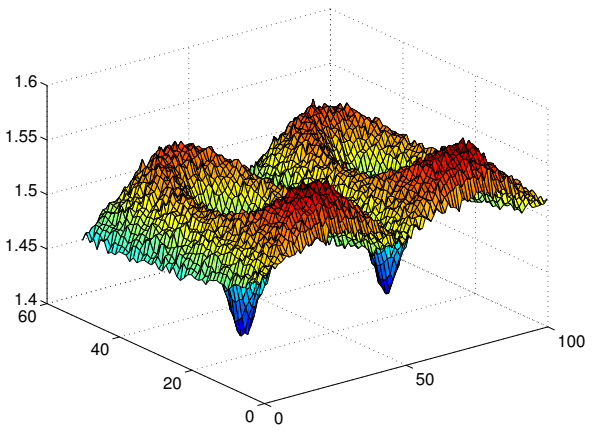


Figure 10.2.: Scattering from a V-sample, measured by a spherical PSD. The sphere has been transformed onto a plane and the intensity is plotted as the third dimension.

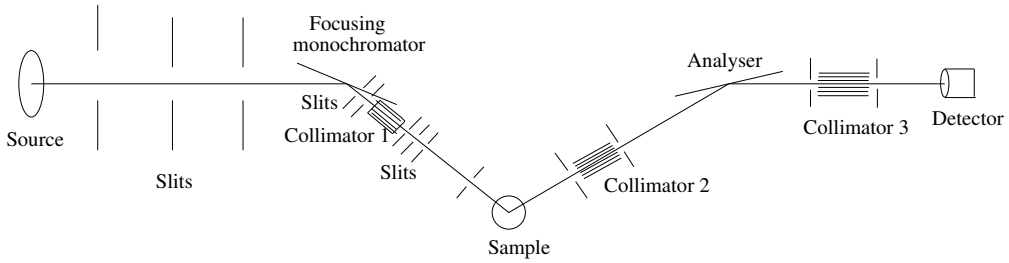


Figure 10.3.: A sketch of the TAS1 instrument.

The analyser of the spectrometer is flat and made from pyrolytic graphite. It is placed between an entry and an exit channel, the latter leading to a single detector. All this has been copied into the instrument definition.

On the spectrometer, Soller collimators may be inserted at three positions: Between monochromator and sample, between sample and analyser, and between analyser and detector. In our instrument definition, we have used 30', 28', and 67' collimators on these three positions, respectively.

An illustration of the TAS1 instrument is shown in figure 10.3. Test results and data from the real spectrometer are shown in Appendix 10.3.1.

10.3.1. Simulated and measured resolution of TAS1

In order to test the McStas package on a qualitative level, we have performed a very detailed comparison of a simulation with a standard experiment from TAS1. The measurement series constitutes a complete alignment of the spectrometer, using the direct beam and scattering from V and Al_2O_3 samples at an incoming energy of 20.0 meV, using the second order scattering from the monochromator.

In these simulations, we have tried to reproduce every alignment scan with respect to position and width of the peaks, whereas we have not tried to compare absolute intensities. Below, we show a few comparisons of the simulations and the measurements.

Figure 10.4 shows a scan of $2\theta_m$ on the collimated direct beam in two-axis mode. A 1 mm slit is placed on the sample position. Both the measured width and non-Gaussian peak shape are well reproduced by the McStas simulations.

In contrast, a simulated $2\theta_a$ scan in triple-axis mode on a V-sample showed a surprising offset from 0 degrees. However, a simulation with a PSD on the sample position showed that the beam center was 1.5 mm off from the center of the sample, and this was important since the beam was no wider than the sample itself. A subsequent centering of the beam resulted in a nice agreement between simulation and measurements. For a comparison on a slightly different instrument (analyser-detector collimator inserted), see Figure 10.5.

The result of a $2\theta_s$ scan on an Al_2O_3 powder sample in two-axis mode is shown in Figure 10.6. Both for the scan in focusing mode $(+ - +)$ and for the one in defocusing mode $(+ + +)$ (not shown), the agreement between simulation and experiment is excellent.

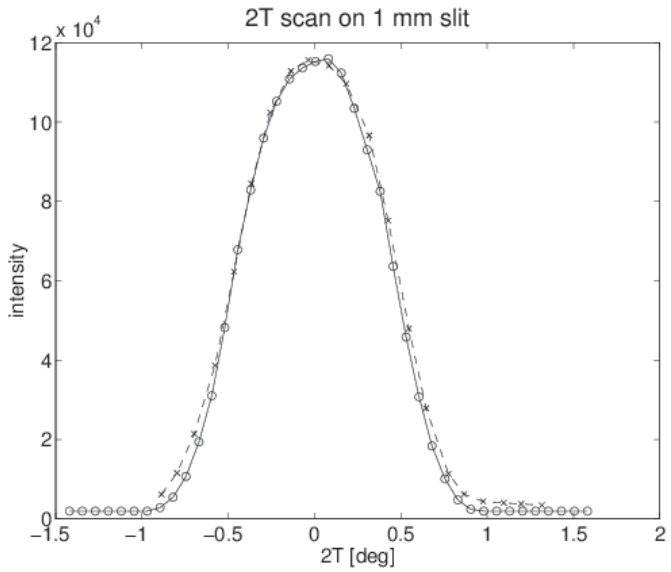


Figure 10.4.: TAS1: Scans of $2\theta_s$ in the direct beam with 1 mm slit on the sample position. "×": measurements, "o": simulations, scaled to the same intensity Collimations: open-30'-open-open.

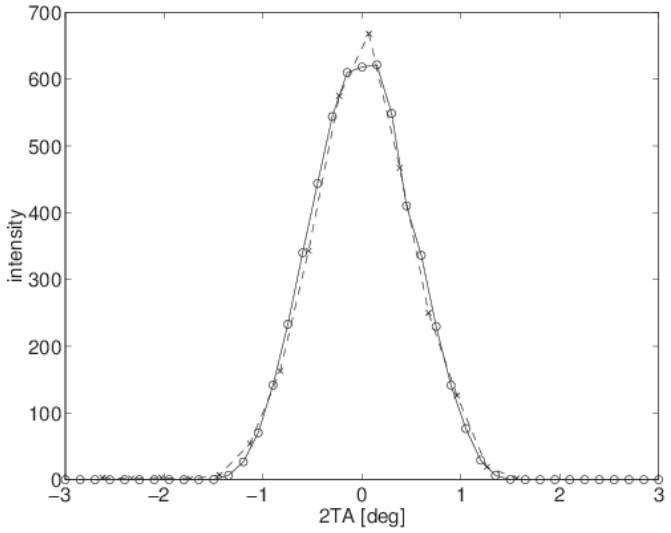


Figure 10.5.: TAS1: Corrected $2\theta_a$ scan on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

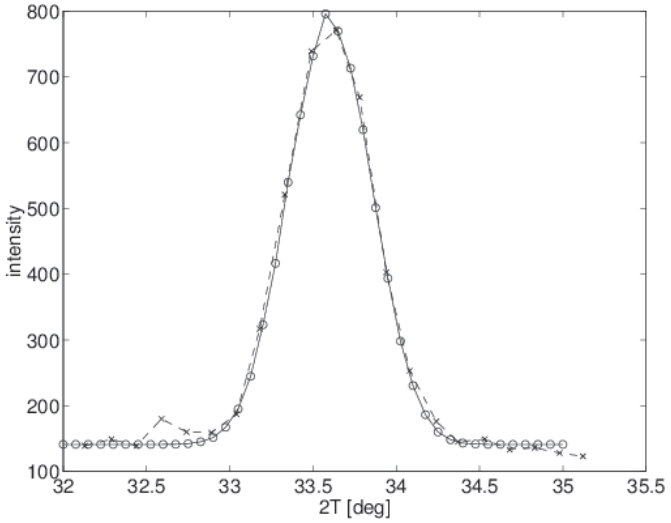


Figure 10.6.: TAS1: $2\theta_s$ scans on Al_2O_3 in two-axis, focusing mode. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations. A constant background is added to the simulated data.

As a final result, we present a scan of the energy transfer $E_a = \hbar\omega$ on a V-sample. The data are shown in Figure 10.7.

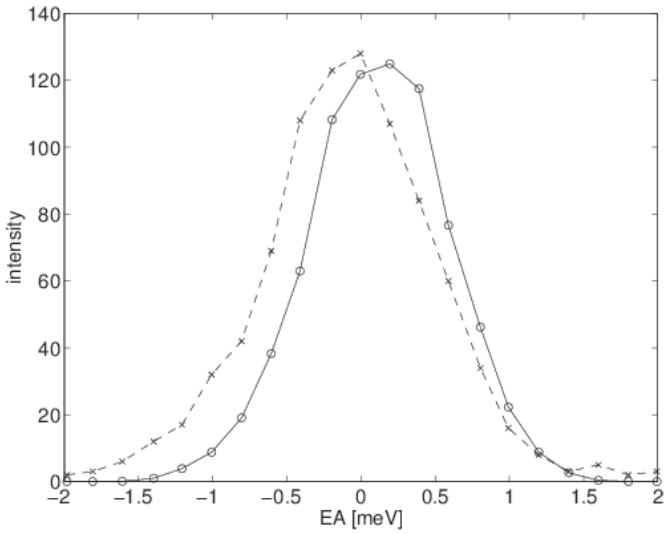


Figure 10.7.: TAS1: Scans of the analyser energy on a V-sample. Collimations: open-30'-28'-67'. "×": measurements, "o": simulations.

For a modern triple-axis instrument that is not tied to a specific, decommissioned

facility, see the `Templates/templateTAS` example instead, which additionally includes a basic in-plane UB-matrix transformation useful for estimating resolution functions of an arbitrary TAS configuration (see also `mcresplot`, section 4.5.7).

10.4. The time-of-flight spectrometer PRISMA

In order to test the time-of-flight aspect of McStas, we have in collaboration with Mark Hagen, now at SNS, written a simple simulation of a time-of-flight instrument loosely based on the ISIS spectrometer PRISMA. The simulation was used to investigate the effect of using a RITA-style analyser instead of the normal PRISMA backend. The instrument is distributed as `ISIS/ISIS_Prisma2` in the example library.

We have used the simple time-of-flight source **Tof_source**. The neutrons pass through a beam channel and scatter off from a vanadium sample, pass through a collimator on to the analyser. The RITA-style analyser consists of seven analyser crystals that can be rotated independently around a vertical axis. After the analysers we have placed a PSD and a time-of-flight detector.

To illustrate some of the things that can be done in a simulation as opposed to a real-life experiment, this example instrument further discriminates between the scattering off each individual analyser crystal when the neutron hits the detector. The analyser component is modified so that a global variable `neu_color` registers which crystal scatters the neutron ray. The detector component is then modified to construct seven different time-of-flight histograms, one for each crystal (see the source code for the instrument for details). One way to think of this is that the analyser blades paint a color on each neutron which is then observed in the detector. Since McStas 3.x, the same effect could alternatively be achieved more simply using a `USERVARS` field (section 5.3.6) rather than a global `DECLARE` variable, which additionally makes the technique safe under SPLIT/GPU execution. An illustration of the instrument is shown in figure 10.8. Test results are shown in Appendix 10.4.1.

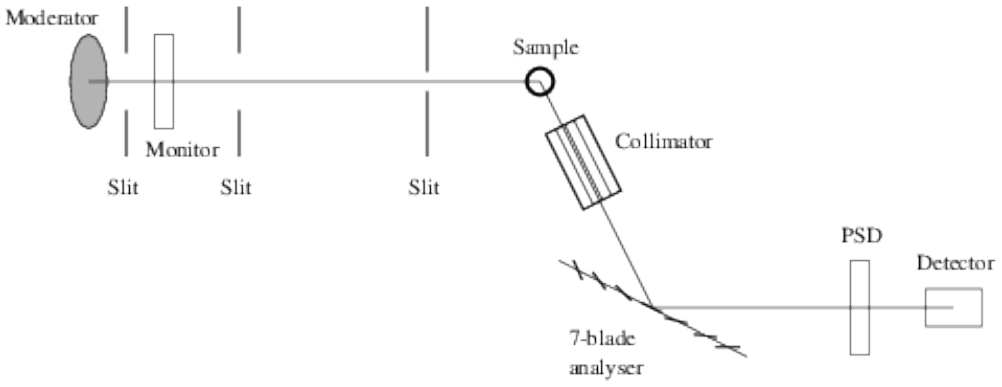


Figure 10.8.: A sketch of the PRISMA instrument.

10.4.1. Simple spectra from the PRISMA instrument

A plot from the detector in the PRISMA simulation is shown in Figure 10.9. These results were obtained with each analyser blade rotated one degree relative to the previous one. The separation of the spectra of the different analyser blades is caused by different energy of scattered neutrons and different flight path length from source to detector. We have not performed any quantitative analysis of the data.

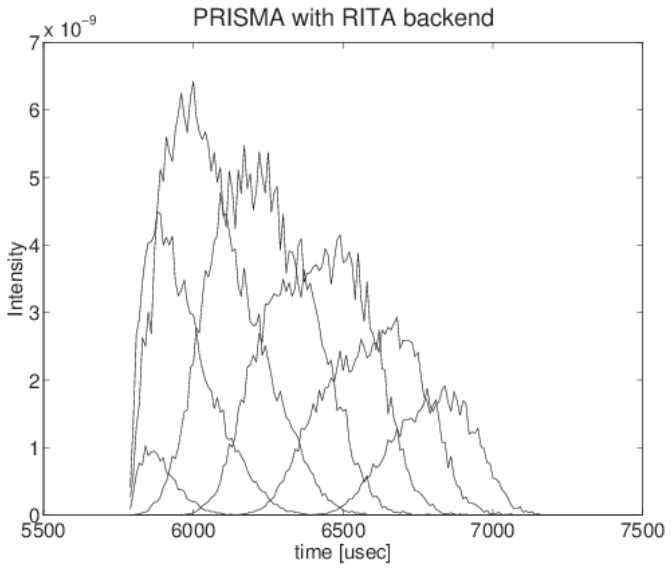


Figure 10.9.: Test result from PRISMA instrument using "colored neutrons". Each graph shows the neutrons scattered from one analyser blade.

A. Random numbers in McStas

A.1. Transformation of random numbers

In order to perform the Monte Carlo choices, one needs to be able to pick a random number from a given distribution. However, most random number generators only give a uniform distribution over a certain interval. We thus need to be able to transform between probability distributions, and we here give a short explanation on how to do this.

Assume that we pick a random number, x, from a distribution $\phi(x)$. We are now interested in the shape of the distribution, $\Psi(y)$, of the transformed $y = f(x)$, assuming $f(x)$ is monotonous. All random numbers lying in the interval $[x; x+dx]$ are transformed to lie within the interval $[y; y + f'(x)dx]$, so the resulting distribution must be $\Psi(y) = \phi(x)/f'(x)$.

If the random number generator selects numbers uniformly in the interval $[0; 1]$, we have $\phi(x) = 1$ (inside the interval; zero outside), and we reach

$$\Psi(y) = \frac{1}{f'(x)} = \frac{d}{dy} f^{-1}(y). \tag{A.1}$$

By indefinite integration we reach

$$\int \Psi(y)dy = f^{-1}(y) = x, \tag{A.2}$$

which is the essential formula for random number transformation, since we in general know $\Psi(y)$ and like to determine the relation $y = f(x)$. Let us illustrate with a few examples of transformations relevant for the McStas components.

The circle For finding a random point within the circle of radius R, one would like to choose the polar angle, ϕ, from a uniform distribution in $[0; 2\pi]$, giving $\Psi_\phi = 1/(2\pi)$. and the radius from the (normalised) distribution $\Psi_r = 2r/R^2$.

For the radial part, eg. (A.2) becomes $y/(2\pi) = x$, whence ϕ is found simply by multiplying a random number (x) with 2π.

For the radial part, the left side of eq. (A.2), gives $\int \Psi(r)dr = \int 2r/R^2 dr = r^2/R^2$, which from (A.2) should equal x. Hence we reach the wanted transformation $r = R\sqrt{x}$.

The sphere For finding a random point on the surface of the unit sphere, we need to determine the two angles, (θ, ϕ).

Ψ_ϕ is chosen from a uniform distribution in $[0; 2\pi]$, giving $\phi = 2\pi x$ as for the circle.

The probability distribution of θ should be $\Psi_\theta = \sin(\theta)$ (for $\theta \in [0; \pi]$), whence by eq. (A.2) $\theta = \cos^{-1}(x)$.

Exponential decay In a simple time-of-flight source, the neutron flux decays exponentially after the initial activation at $t = 0$. We thus want to pick an initial neutron emission time from the normalised distribution $\Psi(t) = \exp(-t/\tau)/\tau$. Use of Eq. (A.2) gives $x = 1 - \exp(-t/\tau)$. For convenience we now use the random variable $x_1 = 1 - x$ (with the same distributions as x), giving the simple expression $t = -\tau \ln(x_1)$.

Normal distributions The important normal distribution can not be reached as a simple transformation of a uniform distribution. In stead, we rely on a specific algorithm for selecting random numbers with this distribution.

A.2. Random generator

As of McStas 3.x, the default random number generator is a 64-bit adaptation of Marsaglia's *KISS* ("Keep It Simple Stupid") generator, a composite of a linear congruential generator, a xorshift generator, and a multiply-with-carry generator, combined to give a very long period ($> 2^{127}$) while remaining fast and simple. Crucially, KISS runs identically on both CPU and GPU/OpenACC, which is why it replaced the previously used Mersenne Twister as the default: Mersenne Twister's large internal state does not parallelize well and is currently not functional for GPU-accelerated simulations (section 4.5.4).

The classic "Mersenne Twister" generator, by Makoto Matsumoto and Takuji Nishimura (see `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` for the original source), remains available as a build-time alternative for CPU-only simulations, selected by compiling the generated instrument with `-DRNG_ALG=1` (e.g. via the `MCSTAS_CFLAGS` environment variable, or `mcstas`'s C-compiler flag options). It has an extremely large period ($2^{19937} - 1$) and negligible cross-correlations up to 623 dimensions, but should not be selected for GPU/OpenACC runs.

B. Libraries and constants

The McStas Library contains a number of built-in functions and conversion constants which are useful when constructing components. These are stored in the `share` directory of the `MCSTAS` library.

Within these functions, the 'Run-time' part is available for all component/instrument descriptions. The other parts are dynamic, that is they are not pre-loaded, but only imported once when a component requests it using the `%include` McStas keyword. For instance, within a component C code block, (usually SHARE or DECLARE):

```
1    %include "read_table-lib"
```

will include the 'read_table-lib.h' file, and the 'read_table-lib.c' (unless the `--no-runtime` option is used with `mcstas`). Similarly,

```
1    %include "read_table-lib.h"
```

will *only* include the 'read_table-lib.h'. The library embedding is done only once for all components (like the SHARE section). For an example of implementation, see **Res_monitor**.

In this Appendix, we present a short list of both each of the library contents and the run-time features.

B.1. Run-time calls and functions (`mcstas-r`)

Here we list a number of preprogrammed macros which may ease the task of writing component and instrument definitions.

B.1.1. Neutron propagation

Propagation routines perform all necessary operations to transport neutron rays from one point to an other. Except when using the special `ALLOW_BACKPROP;` call prior to executing any `PROP_*` propagation, the neutron rays which have negative propagation times are removed automatically.

- **ABSORB**. This macro issues an order to the overall McStas simulator to interrupt the simulation of the current neutron history and to start a new one.
- **PROP_Z0**. Propagates the neutron to the $z = 0$ plane, by adjusting (x, y, z) and t accordingly from knowledge of the neutron velocity (vx, vy, vz). If the propagation time is negative, the neutron ray is absorbed, except if a `ALLOW_BACKPROP;` preceeds it.

For components that are centered along the z-axis, use the `_intersect` functions to determine intersection time(s), and then a `PROP_DT` call.

- **PROP_DT**(dt). Propagates the neutron through the time interval dt, adjusting (x, y, z) and t accordingly from knowledge of the neutron velocity. This macro automatically calls PROP_GRAV_DT when the `--gravitation` option has been set for the whole simulation.

- **PROP_GRAV_DT**(dt, Ax, Ay, Az). Like **PROP_DT**, but it also includes gravity using the acceleration (Ax, Ay, Az). In addition to adjusting (x, y, z) and t, also (vx, vy, vz) is modified.

- **PROP_NO_G_DT**(dt). Propagates the neutron through the time interval dt using straight-line motion, regardless of the global gravitation setting. The negative-time and back-propagation rules are the same as for **PROP_DT**.

- **PROP_NO_G_X0**, **PROP_NO_G_Y0**, and **PROP_NO_G_Z0**. Propagate the neutron to the corresponding local coordinate plane using straight-line motion, regardless of the global gravitation setting.

- **ALLOW_BACKPROP**. Indicates that the next propagation routine will not remove the neutron ray, even if negative propagation times are found. Subsequent propagations are not affected.

- **SCATTER**. This macro is used to denote a scattering event inside a component. It should be used e.g to indicate that a component has interacted with the neutron ray (e.g. scattered or detected). This does not affect the simulation (see, however, **Beamstop**), and it is mainly used by the `MCDISPLAY` section and the `GROUP` modifier See also the SCATTERED variable (below).

B.1.2. Coordinate and component variable retrieval

- **MC_GETPAR**(*comp*, *outpar*). This may be used in e.g. the FINALLY section of an instrument definition to reference the output parameters of a component.

- **NAME_CURRENT_COMP** gives the name of the current component as a string.

- **POS_A_CURRENT_COMP** gives the absolute position of the current component. A component of the vector is referred to as POS_A_CURRENT_COMP.i where i is x, y or z.

- **ROT_A_CURRENT_COMP** and **ROT_R_CURRENT_COMP** give the orientation of the current component as rotation matrices (absolute orientation and the orientation relative to the previous component, respectively). A component of a rotation matrix is referred to as ROT_A_CURRENT_COMP[m][n], where m and n are 0, 1, or 2 standing for x, y and z coordinates respectively.

- **POS_A_COMP**(*comp*) gives the absolute position of the component with the name *comp*. Note that *comp* is not given as a string. A component of the vector is referred to as POS_A_COMP(*comp*).i where i is x, y or z.

- **ROT_A_COMP**(*comp*) and **ROT_R_COMP**(*comp*) give the orientation of the component *comp* as rotation matrices (absolute orientation and the orientation relative to its previous component, respectively). Note that *comp* is not given as a string. A component of a rotation matrice is referred to as ROT_A_COMP(*comp*)[m][n], where m and n are 0, 1, or 2.

- **INDEX_CURRENT_COMP** is the number (index) of the current component (starting from 1).

- **POS_A_COMP_INDEX**(*index*) is the absolute position of component *index*. POS_A_COMP_INDEX (INDEX_CURRENT_COMP) is the same as POS_A_CURRENT_COMP. You may use POS_A_COMP_INDEX (INDEX_CURRENT_COMP+1) to make, for instance, your component access the position of the next component (this is usefull for automatic targeting). A component of the vector is referred to as POS_A_COMP_INDEX(*index*).i where i is x, y or z.

- **POS_R_COMP_INDEX** works the same as above, but with relative coordinates.

- **STORE_NEUTRON**($index, x, y, z, vx, vy, vz, t, sx, sy, sz, p$) stores the current neutron state in the trace-history table, in local coordinate system. *index* is usually INDEX_CURRENT_COMP. This is automatically done when entering each component of an instrument.

- **RESTORE_NEUTRON**($index, x, y, z, vx, vy, vz, t, sx, sy, sz, p$) restores the neutron state to the one at the input of the component *index*. To ignore a component effect, use RESTORE_NEUTRON (INDEX_CURRENT_COMP, $x, y, z, vx, vy, vz, t, sx, sy, sz, p$) at the end of its TRACE section, or in its EXTEND section. These neutron states are in the local component coordinate systems.

- **SCATTERED** is a variable set to 0 when entering a component, which is incremented each time a SCATTER event occurs. This may be used in the `EXTEND` sections to determine whether the component interacted with the current neutron ray.

- **extend_list**(n, &*arr*, &*len*, *elemsize*). Given an array *arr* with *len* elements each of size *elemsize*, make sure that the array is big enough to hold at least n elements, by extending *arr* and *len* if necessary. Typically used when reading a list of numbers from a data file when the length of the file is not known in advance.

- **mcset_ncount**(n). Sets the number of neutron histories to simulate to n.

- **mcget_ncount**(). Returns the number of neutron histories to simulate (usually set by option `-n`).

- **mcget_run_num**(). Returns the number of neutron histories that have been simulated until now.

B.1.3. Coordinate transformations

- **coords_set**(x, y, z) returns a Coord structure (like POS_A_CURRENT_COMP) with x, y and z members.

- **coords_get**$(P, \&x, \&y, \&z)$ copies the x, y and z members of the Coord structure P into x, y, z variables.

- **coords_add**(a, b), **coords_sub**(a, b), **coords_neg**(a) enable to operate on coordinates, and return the resulting Coord structure.

- **rot_set_rotation**(*Rotation t*, ϕ_x, ϕ_y, ϕ_z) Get transformation matrix for rotation first ϕ_x around x axis, then ϕ_y around y, and last ϕ_z around z. t should be a 'Rotation' ([3][3] 'double' matrix).

- **rot_mul***(Rotation t1, Rotation t2, Rotation t3)* performs $t3 = t1.t2$.

- **rot_copy***(Rotation dest, Rotation src)* performs $dest = src$ for Rotation arrays.

- **rot_transpose***(Rotation src, Rotation dest)* performs $dest = src^t$.

- **rot_apply***(Rotation t, Coords a)* returns a Coord structure which is $t.a$

B.1.4. Mathematical routines

- **NORM**(x, y, z). Normalizes the vector (x, y, z) to have length 1 *.

- **scalar_prod**$(a_x, a_y, a_z, b_x, b_y, b_z)$. Returns the scalar product of the two vectors (a_x, a_y, a_z) and (b_x, b_y, b_z).

- **vec_prod**$(a_x,a_y,a_z, b_x,b_y,b_z, c_x,c_y,c_z)$. Sets (a_x, a_y, a_z) equal to the vector product $(b_x, b_y, b_z) \times (c_x, c_y, c_z)$ *.

- **rotate**$(x,y,z, v_x,v_y,v_z, \varphi, a_x,a_y,a_z)$. Set (x, y, z) to the result of rotating the vector (v_x, v_y, v_z) the angle φ (in radians) around the vector (a_x, a_y, a_z) *.

- **normal_vec**(n_x, n_y, n_z, x, y, z). Computes a unit vector (n_x, n_y, n_z) normal to the vector (x, y, z).*

- **solve_2nd_order**$(\&t_1, \&t_2\ A, B, C)$. Solves the 2^{nd} order equation $At^2 + Bt + C = 0$ and returns the solutions into pointers $*t_1$ and $*t_2$. if t_2 is specified as `NULL`, only the smallest positive solution is returned in t_1.

(* The experienced c-programmer may be puzzled that these routines can return information without the use of *pass by reference*, the reason is that these calls are implemented as macros / `#define` wrapped functions.)

B.1.5. Output from detectors

Details about using these functions are given in the McStas User Manual.

- **DETECTOR_OUT_0D**(...). Used to output the results from a single detector. The name of the detector is output together with the simulated intensity and estimated statistical error. The output is produced in a format that can be read by McStas front-end programs.

- **DETECTOR_OUT_1D**(...). Used to output the results from a one-dimensional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

- **DETECTOR_OUT_2D**(...). Used to output the results from a two-dimentional detector. Integrated intensities error etc. is also reported as for DETECTOR_OUT_0D.

- **mcinfo_simulation***(FILE *f, mcformat, char *pre, char *name)* is used to append the simulation parameters into file f (see for instance **Res_monitor**). Internal variable $mcformat$ should be used as specified. Please contact the authors for further information.

B.1.6. Ray-geometry intersections

- **inside_rectangle**(x, y, xw, yh). Return 1 if $-xw/2 \leq x \leq xw/2$ AND $-yh/2 \leq y \leq yh/2$. Else return 0.

- **box_intersect**($\&t_1$, $\&t_2$, x, y, z, v_x, v_y, v_z, d_x, d_y, d_z). Calculates the (0, 1, or 2) intersections between the neutron path and a box of dimensions d_x, d_y, and d_z, centered at the origin for a neutron with the parameters (x, y, z, v_x, v_y, v_z). The times of intersection are returned in the variables t_1 and t_2, with $t_1 < t_2$. In the case of less than two intersections, t_1 (and possibly t_2) are set to zero. The function returns true if the neutron intersects the box, false otherwise.

- **cylinder_intersect**($\&t_1$, $\&t_2$, x, y, z, v_x, v_y, v_z, r, h). Similar to **box_intersect**, but using a cylinder of height h and radius r, centered at the origin.

- **sphere_intersect**($\&t_1$, $\&t_2$, x, y, z, v_x, v_y, v_z, r). Similar to **box_intersect**, but using a sphere of radius r.

B.1.7. Random numbers

- **rand01**(). Returns a random number distributed uniformly between 0 and 1.

- **randnorm**(). Returns a random number from a normal distribution centered around 0 and with $\sigma = 1$. The algorithm used to sample the normal distribution is explained in Ref. [Pre+86, ch.7].

- **randpm1**(). Returns a random number distributed uniformly between -1 and 1.

- **randtriangle**(). Returns a random number from a triangular distribution between -1 and 1.

- **randvec_target_circle**($\&v_x$, $\&v_y$, $\&v_z$, $\&d\Omega$, aim_x, aim_y, aim_z, r_f). Generates a random vector (v_x, v_y, v_z), of the same length as (aim_x, aim_y, aim_z), which is targeted at a *disk* centered at (aim_x, aim_y, aim_z) with radius r_f (in meters), and perpendicular to the *aim* vector.. All directions that intersect the circle are chosen with equal probability. The solid angle of the circle as seen from the position of the neutron is returned in $d\Omega$. This routine was previously called **randvec_target_sphere** (which still works).

- **randvec_target_rect_angular**($\&v_x, \&v_y, \&v_z, \&d\Omega, \mathrm{aim}_x, \mathrm{aim}_y, \mathrm{aim}_z, h, w, Rot$) does the same as randvec_target_circle but targetting at a rectangle with angular dimensions h and w (in **radians**, not in degrees as other angles). The rotation matrix *Rot* is the coordinate system orientation in the absolute frame, usually ROT_A_CURRENT_COMP.

- **randvec_target_rect**($\&v_x, \&v_y, \&v_z, \&d\Omega, \mathrm{aim}_x, \mathrm{aim}_y, \mathrm{aim}_z, height, width, Rot$) is the same as randvec_target_rect_angular but *height* and *width* dimensions are given in meters. This function is useful to e.g. target at a guide entry window or analyzer blade.

B.2. Reading a data file into a vector/matrix (Table input, `read_table-lib`)

The `read_table-lib` provides functionalities for reading text (and binary) data files. To use this library, add a `%include "read_table-lib"` in your component definition DECLARE or SHARE section. Tables are structures of type `t_Table` (see `read_table-lib.h` file for details):

```
/* t_Table structure (most important members) */
double *data;      /* Use Table_Index(Table, i j) to get element [i,j] */
long    rows;      /* number of rows */
long    columns;   /* number of columns */
char   *header;    /* the header with comments */
char   *filename;  /* file name or title */
double  min_x;     /* minimum value of 1st column/vector */
double  max_x;     /* maximum value of 1st column/vector */
```

Available functions to read *a single* vector/matrix are:

- **Table_Init**(&*Table*, *rows*, *columns*) returns an allocated Table structure. Use $rows = columns = 0$ not to allocate memory and return an empty table. Calls to Table_Init are *optional*, since initialization is being performed by other functions already.

- **Table_Read**(&*Table*, *filename*, *block*) reads numerical block number *block* (0 to concatenate all) data from *text* file *filename* into *Table*, which is as well initialized in the process. The block number changes when the numerical data changes its size, or a comment is encoutered (lines starting by '`# ; % /`'). If the data could not be read, then *Table.data* is NULL and $Table.rows = 0$. You may then try to read it using Table_Read_Offset_Binary. Return value is the number of elements read.

- **Table_Read_Offset**(&*Table*, *filename*, *block*, &*offset*, n_{rows}) does the same as Table_Read except that it starts at offset *offset* (0 means begining of file) and reads n_{rows} lines (0 for all). The *offset* is returned as the final offset reached after reading the n_{rows} lines.

- **Table_Read_Offset_Binary**(&*Table*, *filename*, *type*, *block*, &*offset*, n_{rows}, $n_{columns}$) does the same as Table_Read_Offset, but also specifies the *type* of the file (may be "float" or "double"), the number n_{rows} of rows to read, each of them having $n_{columns}$ elements. No text header should be present in the file.

- **Table_Rebin**(&*Table*) rebins all *Table* rows with increasing, evenly spaced first column (index 0), e.g. before using Table_Value. Linear interpolation is performed for all other columns. The number of bins for the rebinned table is determined from the smallest first column step.

- **Table_Info**(*Table*) print information about the table *Table*.

- **Table_Index**($Table, m, n$) reads the $Table[m][n]$ element.

- **Table_Value**($Table, x, n$) looks for the closest x value in the first column (index 0), and extracts in this row the n-th element (starting from 0). The first column is thus the 'x' axis for the data.

- **Table_Free**(&*Table*) free allocated memory blocks.

- **Table_Value2d**($Table, X, Y$) Uses 2D linear interpolation on a Table, from (X,Y) coordinates and returns the corresponding value.

Available functions to read *an array* of vectors/matrices in a *text* file are:

- **Table_Read_Array**(*File*, &n) read and split *file* into as many blocks as necessary and return a `t_Table` array. Each block contains a single vector/matrix. This only works for text files. The number of blocks is put into n.

- **Table_Free_Array**(&*Table*) free the *Table* array.

- **Table_Info_Array**(&*Table*) display information about all data blocks.

The format of text files is free. Lines starting by '`# ; % /`' characters are considered to be comments, and stored in *Table.header*. Data blocks are vectors and matrices. Block numbers are counted starting from 1, and changing when a comment is found, or the column number changes. For instance, the file 'MCSTAS/data/BeO.trm' (Transmission of a Berylium filter) looks like:

```
# BeO transmission, as measured on IN12
# Thickness: 0.05 [m]
# [ k(Angs-1) Transmission (0-1) ]
# wavevector multiply
1.0500   0.74441
1.0750   0.76727
1.1000   0.80680
...
```

Binary files should be of type "float" (i.e. REAL*32) and "double" (i.e. REAL*64), and should *not* contain text header lines. These files are platform dependent (little or big endian).

The *filename* is first searched into the current directory (and all user additional locations specified using the `-I` option, see the 'Running McStas' chapter in the User Manual), and if not found, in the `data` sub-directory of the `MCSTAS` library location. This way, you do not need to have local copies of the McStas Library Data files (see table 9.1).

A usage example for this library part may be:

```
t_Table Table;        // declare a t_Table structure
char file[]="BeO.trm"; // a file name
double x,y;

Table_Read(&Table, file, 1); // initialize and read the first numerical
    block
Table_Info(Table);           // display table informations
...
x = Table_Index(Table, 2,5); // read the 3rd row, 6th column element
                             // of the table. Indexes start at zero in C
                                .
y = Table_Value(Table, 1.45,1); // look for value 1.45 in 1st column (x
    axis)
                             // and extract 2nd column value of that row
Table_Free(&Table);          // free allocated memory for table
```

Additionally, if the block number (3rd) argument of **Table_Read** is 0, all blocks will be concatenated. The **Table_Value** function assumes that the 'x' axis is the first column (index 0). Other functions are used the same way with a few additional parameters, e.g. specifying an offset for reading files, or reading binary data.

This other example for text files shows how to read many data blocks:

```
t_Table *Table;       // declare a t_Table structure array
long     n;
double y;
```

```
4
5   Table = Table_Read_Array("file.dat", &n); // initialize and read the all
        numerical block
6   n = Table_Info_Array(Table);      // display informations for all blocks (
        also returns n)
7
8   y = Table_Index(Table[0], 2,5);   // read in 1st block the 3rd row, 6th
        column element
9                                     // ONLY use Table[i] with i < n !
10  Table_Free_Array(Table);          // free allocated memory for Table
```

You may look into, for instance, the source files for **Monochromator_curved** or **Virtual_input** for other implementation examples.

B.3. Monitor_nD Library

This library gathers a few functions used by a set of monitors e.g. Monitor_nD, Res_monitor, etc. It is also used by **Virtual_output**, a code-specific event-file writer that has moved to the **obsolete** component category (see the Component Manual's Obsolete chapter) and is superseded by **MCPL_output**; new instrument work should prefer MCPL. It may monitor any kind of data, create the data files, and may display many geometries (for `mcdisplay`). Refer to these components for implementation examples, and ask the authors for more details.

B.4. Adaptive importance sampling Library

This library is currently only used by the components **Source_adapt** and **Adapt_check**. It performs adaptive importance sampling of neutrons for simulation efficiency optimization. Refer to these components for implementation examples, and ask the authors for more details.

B.5. Vitess import/export Library

This library is used by the components **Vitess_input** and **Vitess_output**, as well as the `mcstas2vitess` utility. Both **Vitess_input** and **Vitess_output** have moved to the **obsolete** component category (see the Component Manual's Obsolete chapter) and are kept only for backward compatibility; new instrument work should prefer the portable, multi-code MCPL format (**MCPL_input**/**MCPL_output**, see the Misc chapter) over Vitess-specific event files. Refer to these components for implementation examples, and ask the authors for more details.

B.6. Constants for unit conversion etc.

The following predefined constants are useful for conversion between units

Name	Value	Conversion from	Conversion to
DEG2RAD	$2\pi/360$	Degrees	Radians
RAD2DEG	$360/(2\pi)$	Radians	Degrees
MIN2RAD	$2\pi/(360 \cdot 60)$	Minutes of arc	Radians
RAD2MIN	$(360 \cdot 60)/(2\pi)$	Radians	Minutes of arc
V2K	$10^{-10} \cdot m_{\mathrm{N}}/\hbar$	Velocity (m/s)	**k**-vector ($\mathrm{\AA}^{-1}$)
K2V	$10^{10} \cdot \hbar/m_{\mathrm{N}}$	**k**-vector ($\mathrm{\AA}^{-1}$)	Velocity (m/s)
VS2E	$m_{\mathrm{N}}/(2e)$	Velocity squared ($\mathrm{m^2\ s^{-2}}$)	Neutron energy (meV)
SE2V	$\sqrt{2e/m_{\mathrm{N}}}$	Square root of neutron energy ($\mathrm{meV^{1/2}}$)	Velocity (m/s)
FWHM2RMS	$1/\sqrt{8\log(2)}$	Full width half maximum	Root mean square (standard deviation)
RMS2FWHM	$\sqrt{8\log(2)}$	Root mean square (standard deviation)	Full width half maximum
MNEUTRON	$1.67492 \cdot 10^{-27}$ kg	Neutron mass, m_{n}	
HBAR	$1.05459 \cdot 10^{-34}$ Js	Planck constant, $\hbar$	
PI	3.14159265...	π	
FLT_MAX	3.40282347E+38F	a big float value	

C. The McStas terminology

This is a short explanation of phrases and terms which have a specific meaning within McStas. We have tried to keep the list as short as possible with the risk that the reader may occasionally miss an explanation. In this case, you are more than welcome to contact the McStas core team.

- **Arm** A generic McStas component which defines a frame of reference for other components.
- **Component** One unit (*e.g.* optical element) in a neutron spectrometer. These are considered as Types of elements to be instantiated in an Instrument description.
- **Component instance** A named Component (of a given Type) inserted in an Instrument description.
- **Definition parameter** An input parameter for a component. For example the radius of a sample component or the divergence of a collimator.
- **Input parameter** For a component, either a definition parameter or a setting parameter. These parameters are supplied by the user to define the characteristics of the particular instance of the component definition. For an instrument, a parameter that can be changed at simulation run-time.
- **Instrument** An assembly of McStas components defining a neutron spectrometer.
- **Kernel** The McStas meta-language definition and the associated compiler `mcstas`.
- **McStas** Monte Carlo Simulation of Triple Axis Spectrometers (the name of this package). Pronunciation ranges from *mex-tas*, to *mac-stas* and *m-c-stas*.
- **Output parameter** An output parameter for a component. For example the counts in a monitor. An output parameter may be accessed from the instrument in which the component is used using `MC_GETPAR`.
- **Run-time** C code, contained in the files `mcstas-r.c` and `mcstas-r.h` included in the McStas distribution, that declare functions and variables used by the generated simulations.
- **Setting parameter** Similar to a definition parameter, but with the restriction that the value of the parameter must be a number.

Bibliography

[Mcsa] See `https://www.mcstas.org` (cit. on pp. 10, 11, 13, 17, 25, 28, 32, 69, 83, 94).

[Git] See `https://github.com/mccode-dev/McCode/issues` (cit. on pp. 10, 16).

[Nmi] See `http://neutron-eu.net/en` (cit. on p. 10).

[Mcn] See `http://mcnsi.risoe.dk` (cit. on p. 10).

[Ts2] See `http://ts-2.isis.rl.ac.uk` (cit. on p. 10).

[Ess] See `https://www.ess.eu` (cit. on pp. 10, 12).

[Sin] See `http://sine2020.eu` (cit. on p. 10).

[Nis] See `http://strider.lansce.lanl.gov/NISP/Welcome.html` (cit. on p. 11).

[Vit] See `http://www.hmi.de/projects/ess/vitess` (cit. on p. 11).

[Ibw] See `http://neutrons-dev.ornl.gov/eqsans/software/ib/` (cit. on p. 11).

[Nad] `http://code.google.com/p/jnads` (cit. on p. 11).

[Sns] See `http://www.sns.gov` (cit. on p. 12).

[Wik] See `https://github.com/mccode-dev/McCode/wiki` (cit. on pp. 16, 57).

[Mcsb] McStasScript documentation, see `https://panosc-vinyl.github.io/McStasScript/` (cit. on pp. 30, 51).

[Ifi] See `http://ifit.neutroncode.org` (cit. on p. 40).

[Sci] See `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html` (cit. on p. 40).

[Nex] See `http://www.neutron.anl.gov/nexus` (cit. on pp. 60, 72).

[Mcsc] See `https://panosc-vinyl.github.io/McStasScript/index.html` (cit. on p. 124).

[ACL98] A. Abrahamsen, N. B. Christensen, and E. Lauridsen. *McStas simulations of the TAS1 spectrometer*. Student's report. Niels Bohr Institute, University of Copenhagen, 1998 (cit. on p. 135).

[Ali04] L. Alianelli. In: *J. Appl. Cryst.* 37 (2004), p. 732 (cit. on p. 12).

[Ber17] Mads Bertelsen. "Software for simulation and design of neutron scattering instrumentation". PhD thesis. University of Copenhagen, 2017 (cit. on pp. 111, 121).

[BJL17] Mads Bertelsen, Henrik Jacobsen, and Kim Lefmann. "Expanding the McStas sample simulation logic with McStas Union components". In: *Review of Scientific Instruments* 88 (2017), p. 085106. DOI: 10.1063/1.4994697 (cit. on p. 111).

[Cla+66] C.D. Clark et al. In: *J. Sci. Instrum.* 43 (1966), p. 1 (cit. on p. 12).

[Cla+98] K. N. Clausen et al. "The RITA spectrometer at Risø - design considerations and recent results". In: *Physica B* 241-243 (1998), pp. 50–55 (cit. on p. 12).

[Cop+86] J. R. D. Copley et al. "Improved Monte Carlo calculation of multiple scattering effects in thermal neutron scattering experiments". In: *Comput. Phys. Commun.* 40 (1986), p. 337. ISSN: 0010-4655. DOI: http://dx.doi.org/10.1016/0010-4655(86)90118-9 (cit. on p. 13).

[Cop93] J.R.D. Copley. In: *J. Neut. Research* 1 (1993), p. 21 (cit. on pp. 11–13, 19).

[Cop03] J.R.D. Copley. In: *Nucl. Instr. Meth.* A510 (2003), p. 318 (cit. on p. 12).

[Cus03] L. D. Cussen. In: *J. Appl. Cryst.* 36 (2003), p. 1204 (cit. on p. 12).

[EF14] P. Willendrup E. Farhi Y. Debab. "iFit: A new data analysis framework". In: *Journal of Neutron Research* 17.1 (2014), pp. 5–18. ISSN: 1023-8166. DOI: 10.3233/JNR-130001 (cit. on p. 40).

[Far+02] E. Farhi et al. In: *Appl. Phys.* A 74 (2002), S1471 (cit. on pp. 11, 13, 19).

[FMM47] E. Fermi, J. Marshall, and L. Marshall. In: *Phys. Rev.* 72 (1947), p. 193 (cit. on p. 12).

[Fre83] A.K. Freund. In: *Nucl. Instr. Meth.* 213 (1983), p. 495 (cit. on p. 12).

[GRR92] Grimmett, G. R., and Stirzakerand D. R. *Probability and Random Processes, 2nd Edition.* Clarendon Press, Oxford, 1992 (cit. on p. 19).

[Jam80] F. James. In: *Rep. Prog. Phys.* 43 (1980), p. 1145 (cit. on pp. 13, 19, 23).

[KKW14] Erik Bergbäck Knudsen, Esben Bryndt Klinkby, and Peter Kjæ r Willendrup. "McStas event logger: Definition and applications". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 738.0 (2014), pp. 20 –24. ISSN: 0168-9002. DOI: http://dx.doi.org/10.1016/j.nima.2013.11.071 (cit. on p. 128).

[LW02] W.-T. Lee and X.-L. Wang. In: *Neutron News* 13 (2002). See website http://www.sns.gov/ideas/, p. 30 (cit. on pp. 11, 13).

[LN99] K. Lefmann and K. Nielsen. "McStas, a general software package for neutron ray-tracing simulations". In: *Neutron News* 10 (1999), pp. 20–23. DOI: 10.1080/10448639908233684 (cit. on pp. 10, 11, 13).

[Lef+00] K. Lefmann et al. "Added flexibility in triple axis spectrometers: the two RITAs at Risø". In: *Physica B* 283 (2000), pp. 343–354 (cit. on p. 12).

[Lie05] K. Lieutenant. In: *J. Phys.: Condens. Matter* 17 (2005), S167 (cit. on pp. 11, 13, 19).

[Lov84] S.W. Lovesey. *Theory of neutron scattering from condensed matter*. Oxford Clarendon Press, 1984 (cit. on pp. 96, 99, 100, 102).

[Low60] R.A. Lowde. In: *J. Nucl. Energy, Part A: Reactor Sci.* 11 (1960), p. 69 (cit. on p. 12).

[MLS63] H. Maier-Leibnitz and T. Springer. In: *Reactor Sci. Technol.* 17 (1963), p. 217 (cit. on p. 12).

[Man+04] G. Manzin et al. In: *Nucl. Instr. Meth.* A535 (2004), p. 102 (cit. on p. 12).

[Mas+95] T. E. Mason et al. "RITA: The reinvented triple axis spectrometer". In: *Can. J. Phys.* 73 (1995), pp. 697–702 (cit. on p. 12).

[Mil90] D.F.R. Mildner. In: *Nucl. Instr. Meth.* A290 (1990), p. 189 (cit. on p. 12).

[MPC77] D.F.R. Mildner, C.A. Pellizari, and J.M. Carpenter. In: *Acta. Cryst. A* 33 (1977), p. 954 (cit. on p. 13).

[Pes+89] V. Peskov et al. In: *Nucl. Instr. and Meth.* A277 (1989), p. 547 (cit. on p. 12).

[Pet05] J. Peters. In: *Nucl. Instr. Meth.* A540 (2005), p. 419 (cit. on p. 12).

[Pre+86] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1986 (cit. on p. 149).

[Rad74] V. Radeka. In: *IEEE Trans. Nucl. Sci.* NS-21 (1974), p. 51 (cit. on p. 12).

[SK97] J. Saroun and J. Kulda. In: *Physica B* 234 (1997). See website `http://omega.ujf.cas.cz/restrax/`, p. 1102 (cit. on pp. 11, 13).

[Sch+04] C. Schanzer et al. In: *Nucl. Instr. Meth.* A 529 (2004), p. 63 (cit. on pp. 11, 13, 19).

[Sea97] V.F. Sears. In: *Acta Cryst.* A53 (1997), p. 35 (cit. on p. 12).

[SD01] P. A. Seeger and L. L. Daemen. "Numerical Solution of Bloch's Equation for Neutron Spin Precession". In: *Nucl. Instr. Meth.* 457 (2001), p. 338 (cit. on p. 97).

[See+00] P. A. Seeger et al. In: *Physica B* 283 (2000), p. 433 (cit. on pp. 11, 13).

[SST02] G. Shirane, S.M. Shapiro, and J.M. Tranquada. *Neutron Scattering with a Triple-Axis Spectrometer*. Cambridge University Press, 2002 (cit. on p. 12).

[Web20] T. Weber. *Takin 2*. See `https://doi.org/10.5281/zenodo.4117437`. 2020 (cit. on p. 56).

[Wec+00] D. Wechsler et al. In: *Neutron News* 25 (2000), p. 11 (cit. on pp. 11, 13).

[WFL04] P. Willendrup, E. Farhi, and K. Lefmann. In: *Physica B* 350 (2004), p. 735 (cit. on p. 11).

[Wil+14] Peter Kjær Willendrup et al. "McStas: Past, present and future". In: *Journal of Neutron Research* 17.1 (2014), pp. 35–43. ISSN: 1023-8166. DOI: `10.3233/JNR-130004` (cit. on pp. 10, 11, 13).

[Wil88] W. Gavin Williams. *Polarized Neutrons*. Oxford Clarendon Press, 1988 (cit. on pp. 96, 98, 102).

[ZLa04] G. Zsigmond, K. Lieutenant, and S. Manoshin et al. In: *Nucl. Instr. Meth.* A 529 (2004), p. 218 (cit. on pp. 11, 13, 19).

Index

%BUGS (McDoc section), **95**
%D[ESCRIPTION] (McDoc section), **95**
%E[ND] (McDoc section), **95**
%EX[AMPLE] (McDoc section), **95**
%I[DENTIFICATION] (McDoc section), **95**
%IN[STRUMENT_SITE] (McDoc section), **95**
%L[INK] (McDoc section), **95**
%P[ARAMETERS] (McDoc section), **95**
%VALIDATION (McDoc section), **95**
%include (McStas keyword), 30, 67, 76, 144

ABSOLUTE (McStas keyword), 73
ABSORB (C macro, mcstas-r.h), 67, 74, 80, **87**, *144*
Absorption, 74, 87
adapt_tree-lib (library), **152**
Adaptive sampling, 22
ALLOW_BACKPROP (C macro, mcstas-r.h), **87**, *145*
Arm, *154*
AT (McStas keyword), 73
author (McDoc keyword), **95**
Authors, **4**

Backpropagation, 87
box_intersect (C function, mccode-r.c), *148*
BROWSER (environment variable), 57
Bugs
  hard to detect, 14
  integer versus floating-point division, 73
  JUMPs, 81
  MPI, 64
  SPLIT, 82
  system errors, 39
  WHEN vs GROUP, 79

C code
  embedded, 67, 71, 72, 76
C functions, 144
Citing McStas, **10**
Code generation
  options, 29
Comments, 66
Compilation
  failure, 67, 71, 93
  instrument definition, *see* mcstas (compiler)
COMPONENT (McStas keyword), 73
Component, *154*
  declaration (TRACE), 73
  instance, *154*
Constants, **152**
Conversion
  of units, 152
Coordinate
  retrieval functions, 145
  system, 66
coords_add (C function, mccode-r.c), *147*
coords_get (C function, mccode-r.c), *147*
coords_set (C function, mccode-r.c), *147*
COPY (McStas keyword), 78
cylinder_intersect (C function, mccode-r.c), *148*

Data formats, 34, 44, 59, 60, 72, 88
date (McDoc keyword), **95**
DECLARE (McStas keyword), 71, 86
DEFINE COMPONENT (McStas keyword), 83
DEFINE INSTRUMENT (McStas keyword), 69
Definition parameter, *154*
DEFINITION PARAMETERS (McStas keyword), 83
DEG2RAD (constant), **153**
DEPENDENCY (McStas keyword), 70, 85
Description
  McDoc section, **95**
DETECTOR_OUT_0D (C macro, mccode-r.h), **88**, *148*
DETECTOR_OUT_1D (C macro, mccode-r.h), **88**, *148*
DETECTOR_OUT_2D (C macro, mccode-r.h), **89**, *148*
Direction focusing, 22
Documentation

generator, *see* McDoc

EDITOR (environment variable), 43
Embedded C code, *see* C code
END (McStas keyword), 75, 92
Environment variables
  BROWSER, 57
  EDITOR, 43
  MCSTAS_FORMAT, 28, 34, 55, 59
  MCSTAS, 31, 144, 151
Error estimate, 20
Example
  McDoc section, **95**
EXTEND (McStas keyword), 76, 93, 145

File
  search path, 30
FINALLY (McStas keyword), 75, 90
FLT_MAX (constant), **153**
Focusing
  importance sampling, 22
Formats, *see* Data formats
FWHM2RMS (constant), **153**

Graphical User Interface, *see* mcgui
Gravitation, 66
GROUP (McStas keyword), 76, 77, 145
GUI, *see* mcgui

HBAR (constant), **153**
HDF5, *see* NeXus format
HUP (signal), 39

Identification
  McDoc section, **95**
Importance sampling
  direction focusing, 22
INDEX_CURRENT_COMP (C macro, mccode-r.h), *146*
INHERIT (McStas keyword), 93
INITIALIZE (McStas keyword), 72, 87
Input parameter, *154*
inside_rectangle (C function, mcstas-r.c), *148*
Installation, 25
  directory, 31
Installing, 18
Instrument, *154*
Instrument definition, 69, 75
  units, **66**, 70
Instrument site
  McDoc section, **95**
INT (signal), 39, **75**
ITERATE (McStas keyword), 80

JUMP (McStas keyword), 80

K2V (constant), **153**
Kernel, **65–95**, *154*
Keywords, **68**
  McDoc, **95**
kill, **38**

Language, *see* Meta-language
Library, **144**
  adapt_tree-lib, **152**
  Components
    data, 130, 131
  components, 31, 57, 74, **129**
    data, 151
    share, 65, 67, 144
  mccode-r, **144**
  mcstas-r, **144**
  monitor_nd-lib, **152**
  pol-lib, 103
  read_table-lib, 67, **149**
  run-time, 31, 65, 67, **144**
  shared, *see* Library/components/share
  vitess-lib, **152**
Link
  McDoc section, **95**

Mailing lists
  mcstas-users, 10
Mantid
  transfer events and monitor data, 125
MC_GETPAR (C macro, mccode-r.h), 86, *145*
mccode-r (library), **144**
mccoplot (McStas tool), **55**
MCDISPLAY (McStas keyword), 91, 145
mcdisplay (McStas tool), **52**
McDoc, **94–95**
mcdoc (McStas tool), **57**, 94
mcgui (McStas tool), **27–28**, **43**, 54, 55, 94
MCNP, 34
  exchange events, 128
MCPL, 34, 39
mcplot (McStas tool), 28, 35, 44, 47, 50, **54**
mcplotdiff (McStas tool), **55**
mcresplot (McStas tool), **56**

`mcrun` (McStas tool), **49**
`MCSTAS` (environment variable), 31, 144, 151
McStas
  authors, **4**
  citing, **10**
  compiler, *see* `mcstas`
  design, 14
  installation, 25
  mailing lists, 10
  meta-language, *see* Meta-language
  name, 154
  pronunciation, 154
  release scheme, 27
  running from the GUI, 27–28
  structure, 25, 26
`mcstas` (McStas compiler), **29**
`mcstas-jupylab` (McStas tool), **51**
`mcstas-pygen` (McStas tool), 30
mcstas-r (library), **144**
`mcstas2vitess` (McStas tool), 152
`MCSTAS_FORMAT` (environment variable), 28, 34, 55, 59
`mctest` (McStas tool), **57**
`mcviewtest` (McStas tool), **57**
Meta-language, 14, 65
`METADATA` (McStas keyword), 82
Microsoft Windows, *see* Windows
`MIN2RAD` (constant), **153**
`MNEUTRON` (constant), **153**
`modified by` (McDoc keyword), **95**
monitor_nd-lib (library), **152**
Monte Carlo method, 19
  accuracy, 23
  adaptive sampling, 22
  direction focusing, 22
  stratified sampling, 22
MPI, **43**, **60, 61**
  basic usage, 62
  limitations, 64
  mcrun, 62
  performance, 63
MS Windows, *see* Windows
`MYSELF` (McStas keyword), 80

`NAME_CURRENT_COMP` (C macro, mccode-r.h), *145*
Neutron
  absorption, 74, 87
  state (position, velocity, time, spin, weight), **66**
  weight, *see* Weight
Neutron polarisation, *see* Polarisation
`NEXT` (McStas keyword), 80
NeXus, 43, 60
  extension, 72
`NOACC` (McStas keyword), 85
`NORM` (C macro, mccode-r.h), *147*
`normal_vec` (C function, mccode-r.c), *147*

Optimization, 33, **39**
  compiler option, 32
`origin` (McDoc keyword), **95**
Output parameter, *154*
`OUTPUT PARAMETERS` (McStas keyword), 84, 86

Parallel computing, *see* MPI, **60**
Parameters
  definition, 73, 83
  Instruments, 69
  instruments, 33, 34, 49
  McDoc section, **95**
  optimization, 40, 49
  optional, default value, 33, 69, 84
  protecting, *see* Keyword/OUTPUT
  scans, 49
  setting, 73, 83
Path
  file search, 30
`PI` (constant), **153**
Polarisation, **96**
  field functions, 105
  precession algorithm, **103**
  SimpleNumMagnetPrecession, 104
`POS_A_COMP` (C macro, mccode-r.h), *146*
`POS_A_COMP_INDEX` (C macro, mccode-r.h), *146*
`POS_A_CURRENT_COMP` (C macro, mccode-r.h), *145*
`POS_R_COMP_INDEX` (C macro, mccode-r.h), *146*
Preprocessor macros, 144
`PREVIOUS` (McStas keyword), 74, 80
PRISMA, 140
`PROP_DT` (C macro, mcstas-r.h), *145*
`PROP_GRAV_DT` (C macro, mcstas-r.h), *145*
`PROP_NO_G_DT` (C macro, mcstas-r.h), *145*
`PROP_NO_G_X0` (C macro, mcstas-r.h), *145*
`PROP_NO_G_Y0` (C macro, mcstas-r.h), *145*
`PROP_NO_G_Z0` (C macro, mcstas-r.h), *145*

PROP_Z0 (C macro, mcstas-r.h), 87, *144*
Propagation
  backward, 87

RAD2DEG (constant), **153**
RAD2MIN (constant), **153**
rand01 (C function, mccode-r.c), *148*
randnorm (C function, mccode-r.c), *149*
Random numbers
  generator, **143**
  transformations, **142**
randpm1 (C function, mccode-r.c), *149*
randtriangle (C function, mccode-r.c), *149*
randvec_target_circle (C function, mccode-r.c), *149*
randvec_target_rect (C function, mccode-r.c), *149*
randvec_target_rect_angular (C function, mccode-r.c), *149*
read_table-lib (library), **149**
RELATIVE (McStas keyword), 73
Release scheme, 27
Removed neutron events, 145
RESTORE_NEUTRON (C macro, mcstas-r.h), *146*
RMS2FWHM (constant), **153**
ROT_A_COMP (C macro, mccode-r.h), *146*
ROT_A_CURRENT_COMP (C macro, mccode-r.h), *145*
rot_apply (C function, mccode-r.c), *147*
rot_copy (C function, mccode-r.c), *147*
rot_mul (C function, mccode-r.c), *147*
ROT_R_COMP (C macro, mccode-r.h), *146*
ROT_R_CURRENT_COMP (C macro, mccode-r.h), *145*
rot_set_rotation (C function, mccode-r.c), *147*
rot_transpose (C function, mccode-r.c), *147*
rotate (C macro, mccode-r.h), *147*
ROTATED (McStas keyword), 74, 80
Run-time, *154*

Sampling, 22
SAVE (McStas keyword), 74, 88
scalar_prod (C function, mccode-r.c), *147*
SCATTER (C macro, mcstas-r.h), 77, 80, **87**, *145*, 146
SCATTERED (C macro, mccode-r.h), 77, 79, 88, *146*
SE2V (constant), **153**
SEARCH (McStas keyword), 70
Setting parameter, *154*
SETTING PARAMETERS (McStas keyword), 83
SHARE (McStas keyword), 87, 144
SHELL (McStas keyword), 71
SIGHUP, 39
SIGINT, 39, **75**
Signal handler, **38**
  HUP signal, 39
  INT signal, 39, **75**
  TERM signal, 39, **75**
  USR1 signal, 39
  USR2 signal, 38, 39, **75**
SIGTERM, 39, **75**
SIGUSR1, 39
SIGUSR2, 38, 39, **75**
solve_2nd_order (C function, mccode-r.c), *147*
sphere_intersect (C function, mccode-r.c), *148*
Spin-echo, 109
SPLIT (McStas keyword), 81
Statistics
  uncertainty, 20
STORE_NEUTRON (C macro, mcstas-r.h), *146*
Stratified sampling, 22

Table_Free (C function, read_table-lib.c), *150*
Table_Free_Array (C function, read_table-lib.c), *150*
Table_Index (C function, read_table-lib.c), *150*
Table_Info (C function, read_table-lib.c), *150*
Table_Info_Array (C function, read_table-lib.c), *151*
Table_Init (C function, read_table-lib.c), *150*
Table_Read (C function, read_table-lib.c), *150*
Table_Read_Array (C function, read_table-lib.c), *150*
Table_Read_Offset (C function, read_table-lib.c), *150*
Table_Read_Offset_Binary (C function, read_table-lib.c), *150*

`Table_Rebin` (C function, read_table-lib.c), *150*
`Table_Value` (C function, read_table-lib.c), *150*
`Table_Value2d` (C function, read_table-lib.c), *150*
TAS1, 135
`TERM` (signal), 39, **75**
Time-of-flight spectrometer
  PRISMA, 140
Tools, 25
  external, *see* IDL, Matlab, NeXus, Perl, PGPLOT, VRML/OpenGL
  McStas, *see* mcdisplay, mcdoc, mcformat, mcgui, mcplot, mcresplot, mcrun, mcstas, mcstas2vitess
`TRACE` (McStas keyword), 73, 87
Triple-axis spectrometer
  TAS1, 135
Tripoli, 34

Union, **111–124**
  algorithm, 121
  conditionals, 120
  exit volume, 118
  geometry, 116
  loggers, 119
  masks, 118
  master, 118
  materials, 115
  MCViNE, 114
  processes, 113
  surfaces, 119
  tagging, 120
  Union_init, 113
  Union_stop, 113
Units, **66**
  constants and conversions, **152**
`USERVARS` (McStas keyword), 71
`USR1` (signal), 39
`USR2` (signal), 38, 39, **75**

`V2K` (constant), **153**
Variance, 20
Variance reduction, 22
`vec_prod` (C function, mccode-r.c), *147*
`version` (McDoc keyword), **95**
Virtual sources, 23
Vitess, 34
vitess-lib (library), **152**
`VS2E` (constant), **153**

Weight, **20**
  statistical uncertainty, 20
  transformation, 21
`WHEN` (McStas keyword), 79, 80
Windows
  file and directory names with spaces, 27
`written by` (McDoc keyword), **95**